NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

PARALLEL IMPLEMENTATIONS OF PERSPECTIVE VIEW GENERATOR RAY TRACING ALGORITHMS

by

John P. Buziak

December, 1995

Thesis Advisor: Morris E. Driels

Approved for public release; distribution is unlimited.

19960415 074 DTIC QUALITY INSPECTED 1

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December, 1995	Master's Thesis

4. TITLE AND SUBTITLE Parallel Implementations of Perspective View Generator Ray Tracing Algorithms	5. FUNDING NUMBERS
6. AUTHOR(S) John Phillip Buziak	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

In developing line of sight target acquisition software two approaches have been explored. The first was an object based system (Young and Whitney). Next, and more recently, has been the implementation of a database driven simulation. The heart of this implementation is the Perspective View Generator (PVG) developed for the US Army by Wolfgang Baer. This implementation suffers from two fundamental shortcomings: low frame rates and unrealistic representation of target vehicles. This paper concentrates on using more powerful, multi-processor work stations to improve frame rates. The paper also addresses possible methods for representing vehicles in this multi-processor environment.

14. SUBJECT TERMS Parallel Perspective View Generator			15. NUMBER OF PAGES 149 16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

Approved for public release; distribution is unlimited.

PARALLEL IMPLEMENTATIONS OF PERSPECTIVE VIEW GENERATOR RAY TRACING ALGORITHMS

John P. Buziak
Lieutenant Commander, United States Navy
B.S., Tulane University, 1982

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN MECHANICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1995

Author: _____
John P. Buziak

Approved by: _____
Morris Driels, Thesis Advisor

_____
Matthew D. Kelleher, Chairman
Department of Mechanical Engineering

ABSTRACT

In developing line of sight target acquisition software two approaches have been explored. The first was an object based system (Young and Whitney). Next, and more recently, has been the implementation of a database driven simulation. The heart of this implementation is the Perspective View Generator (PVG) developed for the US Army by Wolfgang Baer. This implementation suffers from two fundamental shortcomings: low frame rates and unrealistic representation of target vehicles. This paper concentrates on using more powerful, multi-processor work stations to improve frame rates. The paper also addresses possible methods for representing vehicles in this multi-processor environment.

TABLE OF CONTENTS

I. INTRODUCTION 1
- A. OBJECT BASED ENVIRONMENTS AND DATABASE ENVIRONMENTS 1
- B. WOLFGANG BAER'S PERSPECTIVE VIEW GENERATOR 4
- C. ISSUES TO BE RESOLVED REGARDING THE PVG 7

II. ANATOMY OF THE PERSPECTIVE VIEW GENERATOR 11
- A. THE VARIOUS VERSIONS OF THE PVG USED IN TESTING 11
- B. A DETAILED DESCRIPTION OF THE DATABASE 12
- C. PVG-MAIN.C AS WRITTEN BY DR. MORRIS DRIELS 13
- D. PERSPECTIVE VIEW GENERATOR INITIALIZATION PROCEDURES 15
- E. SETTING UP TO START A NEW FRAME 17
- F. PERFORMING THE RAY TRACE 18
- G. GETTING THE IMAGE TO THE SCREEN 21

III. OPTIMIZING THE SERIAL PROGRAM 25
- A. TEST PROCEDURES 25
- B. MODIFICATIONS WHICH HAD NO EFFECT 29
- C. THE HORIZONTAL ZIGZAG 29

IV. PARALLELIZING THE PERSPECTIVE VIEW GENERATOR 31
- A. AN INTRODUCTION TO PARALLEL PROGRAMING 31
- B. THE KEY #pragma LEXICON ELEMENTS, WITH EXAMPLES 34
- C. METHODS OF PARALLELIZING CODE 36
- D. THE PARALLEL VERSIONS OF PVG-MAIN 37
- E. SUMMARY OF FRAME RATES 39
- F. TOPICS FOR FURTHER RESEARCH 41

V. REPRESENTING TARGETS IN THE PERSPECTIVE VIEW GENERATOR 43
- A. PRELIMINARY EFFORTS TO INTRODUCE TARGETS 43
- B. TOPICS FOR FURTHER RESEARCH: A MORE REALISTIC TARGET 47

VI. CONCLUSIONS 53

LIST OF REFERENCES 55

APPENDIX A 57

APPENDIX B 65

A. PVG-MAIN.C (FIRST TEST VERSION) 65
B. PVG-NOMO.C (VERSION 2) 66
C. PVG-NOV.C 67
D. INIT.C 68
E. CALL_MAP.C 69
F. GET_BLOCK.C 70
G. DATAGEN.C 72
H. INIT_WINDOW.C 73
I. MOUSE.C 74
J. GENDDCOS.C 76
K. MATNMUL.C 78
L. VEH2UTM_MATRIX.C 79
M. ZIGZAG.C 81
N. ZIGZAGA.C 84
O. HITGRNDA.C 87
P. GRND_INSERT.C 88
Q. DISPLAY.C 89
R. TARGETA.C 90
S. RESTOREA.C 93
T. TESTFN5G.C 95
U. PVG-MAIN2.C 96
V. PVG-MAIN4.C 98
W. VTEST.C 100

APPENDIX C 103
A. LOG810 103
B. LOG98A 106
C. LOG91 112
D. LOG95(ALIOTH) 114
E. LOG96(ALIOTH) 116
F. LOG98(ALIOTH) 121
G. LOG925(TOBAGO) 124
H. LOG121 (ALGIEBA) 128

APPENDIX D 131
A. TEST PROCEDURE NUMBER ONE 131
B. TEST PROCEDURE NUMBER TWO 135

INITIAL DISTRIBUTION LIST 137

ACKNOWLEDGMENT

The author would like to acknowledge the superb technical assistance provided by Mike McCann and Mathew Koebbe of the Scientific Visualization Lab. Without their help and diligence in upgrading the various systems of the Scientific Visualization Lab this research could not have been accomplished.

The author would also like to thank the Oceanography Department for providing access to their systems for testing.

I. INTRODUCTION

In recent years considerable effort has been expended developing virtual environments. The applications for virtual environments are extensive, ranging from simple entertainment to highly advanced training systems. The military and civilian industry have shown considerable interest in the use of virtual environments in training and education. Virtual environments are highly desirable where the cost of operating equipment for training is expensive, or the risk to personnel and equipment is unacceptably high. These conditions are routinely encountered in branches of the military and many fields in civilian industry.

A. OBJECT BASED ENVIRONMENTS AND DATABASE ENVIRONMENTS

In implementing virtual environments two broadly defined approaches have been pursued: Object Based environments and Database environments. Object based environments begin with mathematically defined geometric objects. A database of the relative positions of all the objects is maintained. From this body of information various aspects of the modeled environment can be determined. Most often the visual aspects of an environment must be determined. More precisely, a visual representation of a scene is obtained, but this is not always the case. In some cases the objective is to determine if a series of events can physically be accomplished, such as a long truck completing a turn around a tight corner.

For visualizations, object based environments may suffer from significant drawbacks. Real life objects must be represented using simple geometric shapes. To accurately render these objects may require an exceptional number of shapes at a wide array of angles. As the number of shapes goes up, the computational overhead involved in tracking the objects,

determining aspects and lines of sight, rises dramatically. The process is analogous to animating a stained glass window.

Figure 1 is the screen capture from a Silicon Graphics Demo program. The objects are represented by simple geometric figures. This is typical of the type of compromises

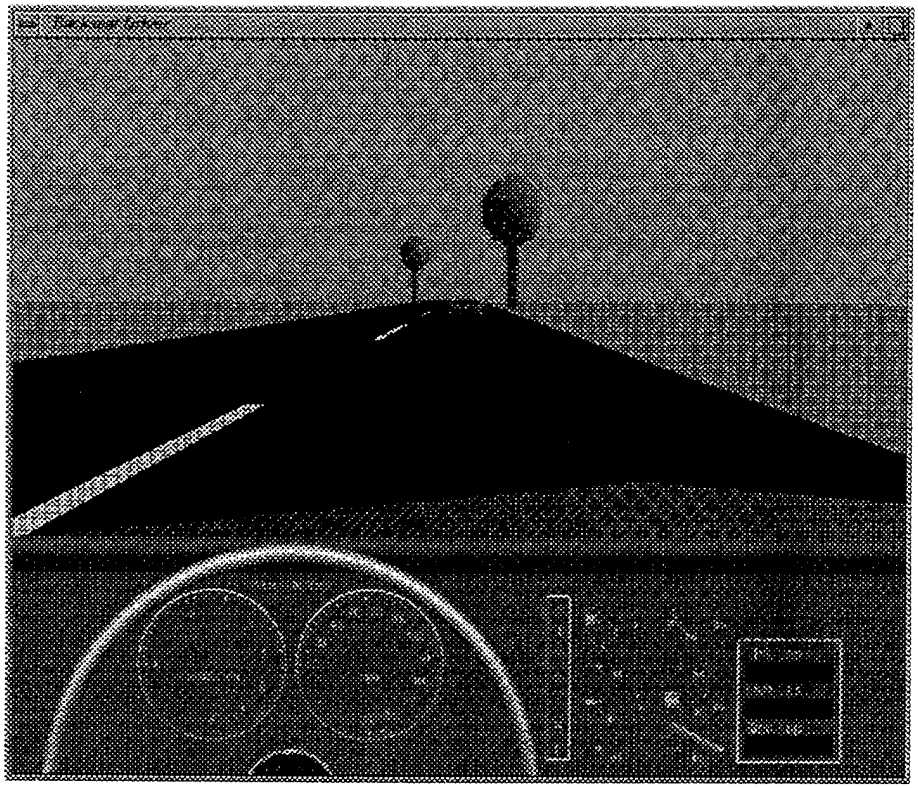


Figure 1 Screen Capture from Silicon Graphics, Inc. Demo.

necessary to achieve continuous motion on a single processor computer architecture. As the tracking over head rises the speed with which scenes can be rendered decreases. This makes the rendering of continuous motion more difficult. Where a balance is struck between speed and the number of shapes used to render an object, the scenes produced have been characterized as "cartoon-like"[Ref. 1:p. 3].

The database approach resembles an attempt to reverse engineer nature. In nature, a ray of light can be traced from a light source to a surface. The reflected ray is then traced to the human eye, or other sensing device (see Figure 2). In the database system a "ray" is traced from the sensing device to a surface (as defined in the database). The color of the surface is returned to the sensing device to make up a portion of the field of view. Due to the method used to render scenes in the database environment, they are often referred to as ray

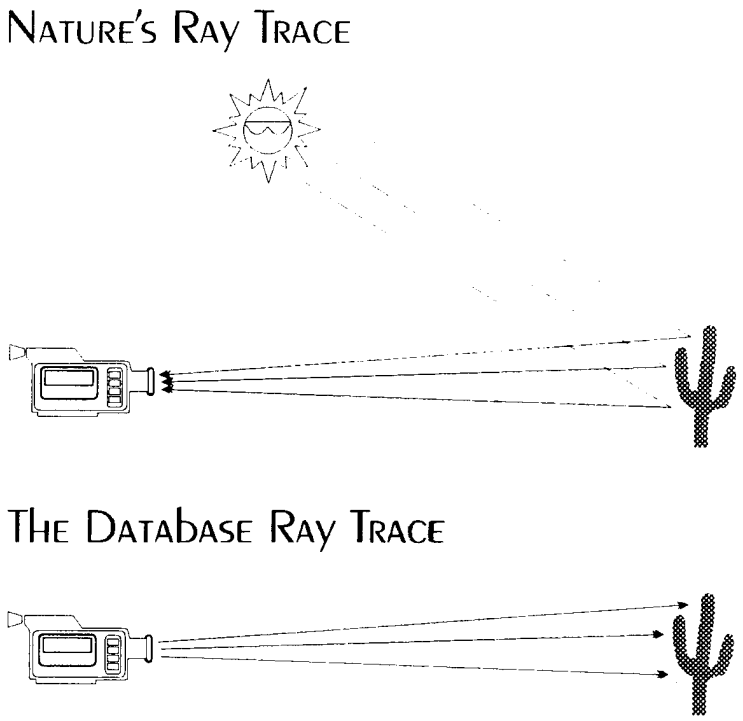


Figure 2 Nature versus the Ray Tracing Algorithm.

tracing systems. As the number of ray traces increases, either the field of view can be expanded, or the resolution improved. It should be noted that there is a point beyond which resolution cannot be improved by adding ray traces. The true limit of the resolution of the image is determined by the resolution of the database.

The database can be thought of as a three dimensional bitmap. The achievable resolution being determined by the portion of the environment being represented by one bit. The bit size can range anywhere from one micron to a parsect. As the number of bits or the number of ray traces increases, the speed with which scenes can be rendered decreases. This approach provides highly accurate representations of the objects in the database at the expense of speed. Consequently, it is extremely difficult to render continuous motion. Although ray tracing is mathematically simple, the same calculation must be repeated hundreds of thousands of times to render a single image of 256 by 256 pixels.

The object of this paper is to present some solutions to problems commonly encountered in implementing ray tracing algorithms. To illustrate these methods a ray tracing system developed for the U.S. Army by Wolfgang Baer will be used. These methods are implementable in any ray tracing algorithm. This is particularly true of the techniques used to implement ray traces on multi-processor computer systems.

B. WOLFGANG BAER'S PERSPECTIVE VIEW GENERATOR

Dr. Baer originally developed his ray tracing algorithm, or Perspective View Generator (PVG) as a training aid for the Fiber Optic Guided Missile system. The database created for this system replicates the terrain found in the Army's training center at Fort Hunter Liggett, California [Ref. 2]. To obtain the three dimensional effect, a two dimensional grid is laid over the area of interest. For each square in the grid the average height of the terrain, including vegetation, is recorded for that position. The grid squares are one meter on a side, and heights are recorded to the nearest meter. Lower resolution databases were also produced, but were not needed for the topics discussed in this thesis.

The overall training system includes an extensive telecommunications suite, with input devices embedded in trainers in the field. For the purposes of this thesis the component of interest is the Stand Alone Perspective View Generator (SOPVG). This device takes data on the position and bore sight of the observer, performs the ray tracings, and produces a corresponding image. The SOPVG accomplishes this using an extensive architecture including 18 transputers. Each transputer could act as a stand alone computer. Employing this architecture, a maximum of nine processors can be operating simultaneously performing ray tracings.[Ref. 2]

In Baer's system the observer is a television camera mounted on a fiber optically guided missile. Although the missile has a limited field of view, it travels at high speeds and could cover a very large area before contacting the target. These conditions drove the requirements for large disk storage and RAM requirements. To achieve the processing speeds desired a pipelined approach in both hardware and software was developed, with parallel elements introduced where possible. This resulted in a system where the hardware and associated software are segregated in to five modules.

At the front end of the pipeline the Data Retrieve Section analyzes the direction of motion of the missile and determines the likely area of the database which will be required to generate the scene. Four transputers, working in parallel, then move this portion of the data base from hard disk storage to RAM storage. Each of the four transputers has 32 megabytes of RAM to support its operations. At this point in the pipeline the SOPVG is also applying the position of the sun to determine the shading on the terrain.[Ref. 2]

In the Terrain Extraction section four transputers are paired with four transputers from the Terrain Rendering Section. The Terrain Extraction section further considers the bore sight of the observer and refines the area required by the ray tracing algorithm. This area is then loaded to twelve modules of 16 megabytes of RAM which are shared with the Terrain Rendering Section.[Ref. 2]

The third step in the pipeline, the Terrain Rendering Section, is where the ray tracing portion known as TER_ZAG is executed. In the TER_ZAG program ray tracing does not start at the observer each time. Instead it takes advantage of the fact that the next ray will contact the terrain at a point very close to the current one. The next ray trace starts directly above the contact point of the current ray's impact point. Hence the "ZAG" in TER_ZAG. This zigzag, or terrain following approach, is one of the keys to rapidly rendering scenes. Without the benefit of the terrain following feature 256x256x500, or just over 32 million steps, would have to be taken to render an image in the SOPVG [Ref. 1: p. 6]. Employing the terrain following feature this number is modified to (256X500)+(255x256), or 195,830 steps. The last two steps in the software pipeline assemble the gray shades in to a scene and manage the video buffer.[Ref 2]

Baer's design is hardware intensive with parallelism accomplished by physically segregating the code. This is a very complex implementation to manage. In the original configuration a frame rate of 3 frames per second was achieved. With hardware upgrades, 5 to 15 frames per second were projected.[Ref. 2] Baer's design is also tied to it's hardware. Transportability to other platforms may be possible, but only with additional development costs.

C. ISSUES TO BE RESOLVED REGARDING THE PVG

One of the primary goals of this thesis was to demonstrate that specialized hardware is not required to implement a perspective view generator. Silicon Graphics machines were selected as target machines due to their ready availability. With an eye on expansion of the PVG concept, the Silicon Graphics machines come in a wide range of architectures suited to various price ranges. They also can accommodate sufficient RAM in a single architecture to permit the entire database be loaded to hard memory. This immediately shortens the pipeline by two steps from the that found in the SOPVG.

In either the object based system or in implementing ray tracing algorithms, the greatest limitation is the speed with which scenes can be rendered. In recent years technology has been generous, providing researchers with faster and more powerful computers. In the course of presenting results in this thesis it will be apparent how the march of technology, by itself, has made ray tracing algorithms more viable. Each passing year breakthroughs which would accelerate individual CPUs become more difficult to achieve. To reach a point where smooth video can be produced from a database other avenues must be explored.

Optimizing the serial code, in this case written in C, offers some small possibilities for improved speed. Most of the opportunities to obtain these types of improvements have already been exploited by Baer, Driels and others.

The most promising approach to "the speed problem" is the segregating of the code to allow the employment of parallel processors. This can be accomplished through various means. The most complicated and difficult to coordinate method is to run identical code on physically separate machines and integrate the results on a processor responsible for control

and coordination. This is the approach implemented in the SOPVG. Another approach is to use "primitives" to parallelize the code and run it on multiple processors within a single architecture. The syntax and implementation of primitives varies in the C language from one computer manufacturer to another. This approach relies heavily on queues. Additionally, as the name implies, primitives are limited in their flexibility and are difficult to work with.

The approach which will be discussed in this thesis is the use of the Power C language on the family of multi-processing systems produced by Silicon Graphics. Rather than a language, Power C is actually a library of compiler directives provided for the C compiler. These compiler directives are actually constructed from the primitives and provide syntax elements which are more readily integrated into a structured programming language. The Power C libraries also provide more methods for constructing parallel code, including code analyzers and optimizers. These will be discussed in more detail later in the thesis.

The availability of the Power C elements permits a more systematic approach to developing code for multi-processors. One well developed and tested approach is provided by Barr E. Bauer in his book Practical Parallel Programing.

- Profile the execution of the serial version of the program.
- Determine the regions in which execution time is significant.
- Look for opportunities in those significant regions.
- Modify dependent code blocks inhibiting parallelization.
- Guard against Dependence.
- Include as much code in the parallel region as possible.

- Examine alternatives.
- Profile again.
- Tune parallel regions.
- Avoid parallelizing:
  All loops.

  Short execution loops. [Ref. 3: p 190]

Not directly related to frame rates, is the problem of handling over hangs. An example of this can be found in Figure 3. The shaded area under the tree is an overhang region. Because of the nature of the ray tracing algorithm and the limited data available from the

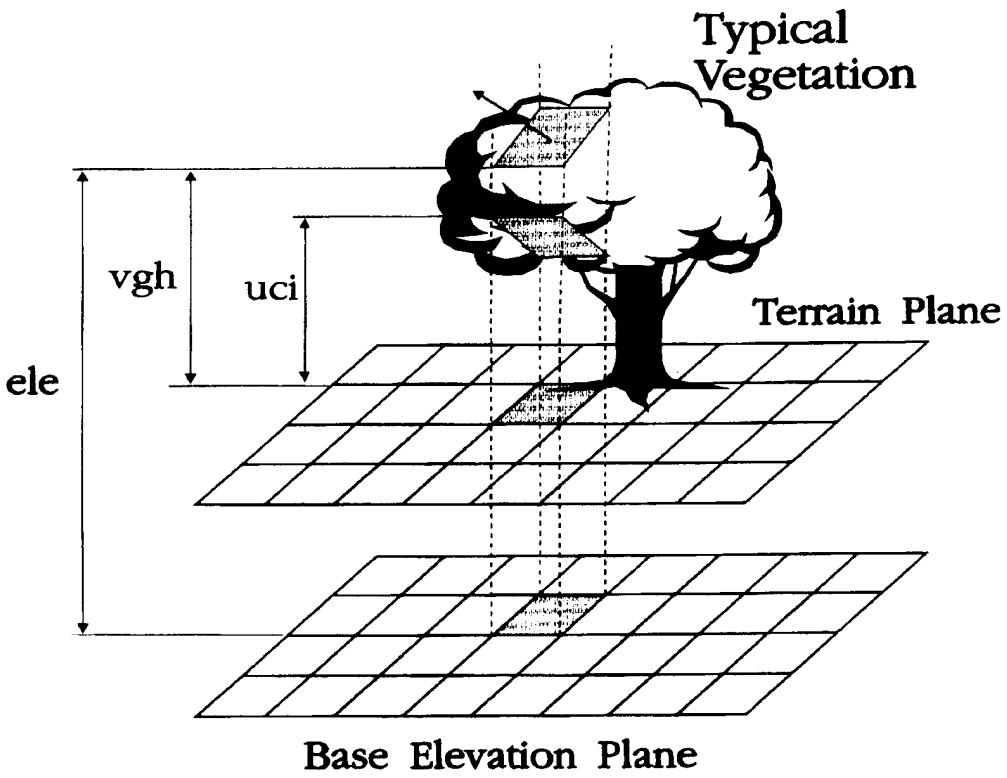


Figure 3 Graphical Representation of Terrain Database Elements.

database, determining when a ray passes underneath a structure can be problematic. One method for resolving this problem without driving up the frame rates will be presented.

Representing target vehicles in a recognizable form is another problem which requires further research. Very few maneuverable man made objects can be reasonably represented using one meter resolution. In the SOPVG, the requirement was to render targets to a resolution of .3 meters[Ref. 2]. The simplest target implementations are conformal, that is they are flexible and follow the contour of the terrain. Vehicles are, of course, rigid. This fact has not yet been accounted for in the algorithms which will be presented here, although recommendations for including targets in the database will be provided.

II. ANATOMY OF THE PERSPECTIVE VIEW GENERATOR

A. THE VARIOUS VERSIONS OF THE PVG USED IN TESTING

At the beginning of testing not all available machines were capable of supporting the same level of graphics. This lead to the development of different versions of the PVG which could allow the performance of the various machines to be compared. The first of these versions is PVG-No Video. This version prints "start" on the screen when scene rendering begins. Once sixty frames have been rendered, "stop" is printed on the screen. Prior to running this series of tests it was believed that the overhead associated with sending the rendered scene to the screen was minimal. This proved not to be the case.

To more precisely determine the overhead involved in writing the image to the screen another version was produced, PVG-Video-no mouse. The position of the cursor on the screen determines the speed and direction of motion of the observer. The mouse is used to position the cursor on the screen. As the name implies, the effects of the mouse were left out. This freezes the position of the observer and effectively freezes the rendered image. As in the version without video, screen writes signal the beginning and end of scene rendering. The impact of video will be more completely discussed in the section of the thesis on results.

The version, PVG-Video with mouse, or PVG-main, is the most complete version of the PVG prior to adding targets. The sole purpose of this version is to determine the frame rate with a moving observer. This can be considered the true frame rate. Beyond this point the addition of targets and other features, will modify the frame rate. The most likely outcome being a reduction in frame rates.

B. A DETAILED DESCRIPTION OF THE DATABASE

A thorough understanding the data base is essential in understanding what can, and more importantly, what cannot be represented in the image sent to the screen. The database employed in the PVG is pseudo three dimensional. Two of the dimensions are completely defined in the Base Elevation Plane (See Figure 3). The third dimension, the vertical one, is partially defined in a 32 bit word. One of these 32 bit words is associated with each box on the Base Elevation Plane. Since each block on the base elevation plane is one meter square, each grid position can be specified using two integer values without any loss of information. The Bit Assignment Table below breaks down the 32 bit word in to it's various components.

3 1	2 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0							
ele	uci	nor	vgh	v i d	n	s	gsv	

Table 1. Bit Assignment Table. After Ref.[2]

- ele - This is the elevation above sea level to the center of the theoretical one meter block above the Base elevation plane.
- uci - The under cover index provides the distance from the overhanging portion of a structure, or vegetation to the Terrain Plane.
- nor - This provides the angle of the surface normal as measured from the vertical. The implementations of the PVG which appear in this thesis do not utilize this data.
- vgh - This is the height of the vegetation measured from the Terrain Plane.

- vid - The vegetation identification is not used in the versions of the PVG discussed in this thesis.
- n - This bit indicates whether the object contained in the block is man made (1) or natural(0).
- s - Indicates whether the surface in question is in direct sun light or shade.
- gsv - gray shade value of the vegetation or bald terrain.

Note that only one gray shade value is retained in the database base for each position on the Base Elevation Plane. Referring to the example in Figure 1 the color retained as the gray shade value for the position indicated by the shading would be the value for the top of the tree. In most situations the gray shade value for area underneath the tree would be substantially different from the tree itself. This difference would be even more acute if seasonal conditions, such as snow cover, are considered.

C. PVG-MAIN.C AS WRITTEN BY DR. MORRIS DRIELS

Development and testing began with a set of prototype programs modeled on the core code used by Wolfgang Baer in the SOPVG. The code was further modified by Dr. Morris Driels to study the PVG's value as a visualization tool for the JANUS(A) gaming system. A flow chart for the main program as conceived by Dr. Driels appears in Appendix A.

The main program can be separated into two distinct sections. Contained between program start and the while loop are a series of functions calls which load the database and initialize the window where the rendered scenes will be displayed. This constitutes the first section, which is executed only once. This portion of the code has no impact on the frame

rate. The second portion of the code is contained within the while loop. Each pass through this while loop represents one frame, or image, being written to the screen. This is the portion of the code where improvements and optimizations will improve the frame rate.

Within the while loop are two function calls which were introduced by Morris Driels to test target acquisition algorithms not related to the issues in this thesis. These are the calls to TARGET and RESTORE. It will be observed in later flow charts and code that these were deleted.

The MOUSE function facilitates movement of the of the observers position. By moving the pointer around the screen the direction of motion of the observer can be changed as well as the speed over the ground. Specific button combinations for maneuvering the observer can be found in the comments at the beginning of the C code for MOUSE (see Appendix B).

In the PVG, rapid ray tracing is accomplished through the use of direction cosines. As the observers position and attitude change with each frame, the direction cosines change. This is accomplished in the function GENDDCOS. The ZIGZAG function takes the direction cosines from GENDDCOS and accomplishes the ray traces and generates an array holding a series gray shade values. The last step in scene rendering is accomplished by the DISPLAY function. DISPLAY takes the array of gray shades produced by ZIGZAG and writes them to the screen as a bit map.

D. PERSPECTIVE VIEW GENERATOR INITIALIZATION PROCEDURES

The function INIT initializes a series of vectors used primarily to describe the starting position of the observer. The vector IFOVNOW consists of ten elements. The first two elements specify the observers position projected down on the Base Elevation Plane. The third element provides the observers elevation. The next three elements specify the bore sight of the observer. The seventh element provides the field of view. The last three elements are not used. RAYSEG provides the dimensions of half of the screen for GENDDCOS. Finally, INIT uses the variables TAR_X and TAR_Y to hold the initial position of a ground target projected on the Base Elevation Plane. These last two variables were not used in the frame rate testing.

The next step in initialization is reading the database from hard disk to the two dimensional array DAT. Before this is be accomplished DATAGEN prompts the user for some basic data. The PVG is capable of displaying over a range of magnifications. With a magnification of one, the window which appears on the screen measures 1.5 inches on a side[Ref. 1]. DATAGEN also requests the user to select a resolution. Several databases are available for Fort Hunter Ligget with varying resolution. For all testing of algorithms discussed in this thesis the one meter resolution database was used. DATAGEN also prompts the user to enter a starting point using the coordinates of the Base Elevation Plane.

Once the preliminaries have been taken care of, DATAGEN calls the function GET_BLOCK to load the database. One 32 bit word is loaded to each of the cells in the array DAT. Each set of indices of the array correspond to a one meter block of the Base

Elevation Plane. The number of blocks loaded is hard coded in the program, but can be adjusted up to the point where the available RAM is exhausted.

Before a scene can be written to a screen, a portion of the screen must be set aside as a window. To accomplish this, two graphics languages are provided by Silicon Graphics. These are GL and OpenGL. The simpler of the two, GL, uses the standard commands provided by the Xwindow interface which comes with standard Unix operating systems to initialize a window. The second, OpenGL, requires a complex and extensive window server be written by the programmer before a window can be opened. The advantage to OpenGL being that once the window has been opened, more powerful and flexible commands are available to manipulate the window.

The GL language was selected for two reasons. The most important of these is the simplicity of programing in GL. The second is somewhat more pragmatic. Three of the four multi-processing computers available for testing are 32 bit machines which only support the GL instruction set. As of August 1995, the 64 bit machine that was available only supported OpenGL. This is the condition which necessitated the development of the PVG-No Video option for testing. This decision was made more acceptable by the prospect that an upgrade to the operating system was in development for the 64 bit machine, which would allow the use of GL code. This upgrade was successfully completed in November 1995.

Graphics language is first encountered in the INIT_WINDOW function. This function opens a window, and modifies it's default settings. The command 'winopen', is fairly self explanatory. It sets aside a portion of the screen to be manipulated by the programmer. The default dimensions of the window are specified prior to opening the window using the

command 'prefsize'[Ref. 4]. Color gives the window a gray shade value to use as a reference, or index[Ref. 4]. RGB allows the color map to be bypassed when rendering scenes[Ref. 4]. Once the window is open none of the default values modified by follow on commands, such as RGB mode, will take effect until the 'gconfig' command is executed[Ref4].

E. SETTING UP TO START A NEW FRAME

The PVG forms a bridge between two distinct reference frames. The first reference plane is that of the observer, referred to in the C code as the 'camera' system. The vector IFOVNOW provides a bore sight that is relative to the primary axes of the observer's reference frame. The other reference frame is the one where most of the work is done in the ray trace. This is the reference plane of the database, referred to in the C code as the 'utm' system . The position of the observer is maintained in the vector IFOVNOW in 'utm' coordinates. This position is updated for each new frame in the function MOUSE, prior to generating the direction cosines for the ray trace.

The means for accomplishing this transformation is well documented in numerous physics and dynamics texts, thus, it will not be repeated here. It is sufficient to note that the transformation requires multiplication of two, three element vectors. This is accomplished by the general purpose function MATNMUL. This process must be repeated three times, for column shifts, row shifts and ray trace steps. To further reduce the redundant code the function VEH2UTM is used to accomplish related mathematics. When the function ends it will have updated the global three by three matrix, ddcos, with the direction cosines necessary to perform the ray trace.

F. PERFORMING THE RAY TRACE

The function ZIGZAG performs the actual ray trace and fills a vector, VIEW, with a gray shade value for each completed ray trace. The flow chart in Appendix A and the code in Appendix B is the baseline function used in all testing to determine frame rates. This function is based on a version written by Morris Driels, portions of which, dealing with vehicle acquisition testing, were removed for frame rate measurements.

The function begins by moving the values for the current positions of the observer in to local variables ('e','n','zr'). The values 'e' and 'n' fix the position of the observer on the Base Elevation plane. The value 'zr' is the height of the observer above the Base Elevation Plane. From this point the direction cosines for GENDDCOS are set up to start ray tracing from the point of the camera to the lower left hand corner of its field of view and further on to the terrain (see Figure 4).

The outermost 'for' loop adjusts the direction cosines at the beginning of each column, shifting the ray from left to right. The next innermost 'do while' loop performs a similar function adjusting the direction of the ray, row by row, trace from the bottom of a column to the top. The inner most loop gets to the actual business of performing the ray trace. This loop successively applies direction cosine to the values of 'e','n' and 'zr'. At each step the height at the indices 'e','n' in the matrix DAT (a.k.a. 'zt') are compared with 'zr'. If the ray height is below the terrain level the function HITGRND is called to determine the gray shade value. The 'if' statement included in this loop guards against the ray running off the edge of the data base. If this should occur the ray trace is terminated using a 'break' command and the pixel for that ray is colored white (gray shade value = 255).

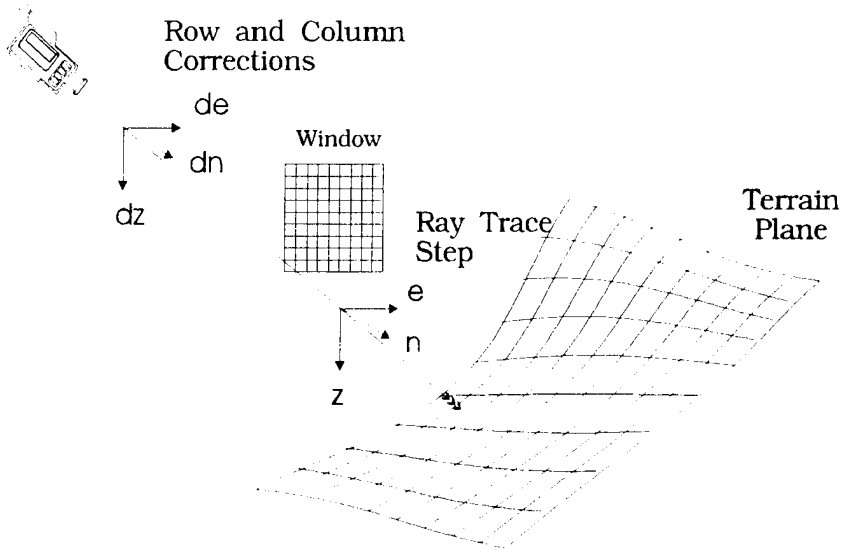


Figure 4 Graphical Relationship of Ray Trace Elements.

The zig-zag pattern, or terrain following feature, is implemented by recording the step where terrain contact was made at the end of the inner ray tracing loop and prior to incrementing the row. The step is then decremented by three to provide the starting point for the next ray trace. This is done to start the ray trace clear of any vegetation overhangs. When starting a new column, the step is set to zero for the row zero trace.

This version of ZIGZAG is also coded to provide a grid reference on the screen in white. This is accomplished in the 'else if' structures encountered on leaving the ray tracing loop.

In determining the gray shade value there are several cases which must be considered. Case A, in Figure 5, shows a ray striking the top of a piece of vegetation. As long as the ray is not below the level of the overhang, then the gray shade value is that of the tree. Case B, shows a slightly lower ray striking the tree on the underside of the overhang. In the current

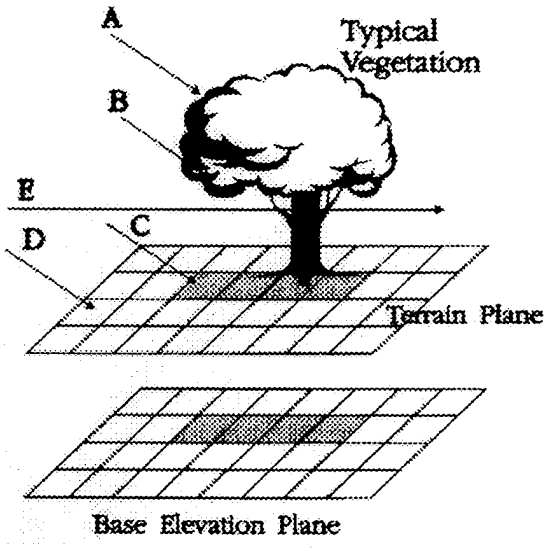


Figure 5 Possible Ray Paths for the HITGRND function.

version of HITGRND the same gray shade value is returned as case A. Depending on the angle of the sun the value returned could be correct, or could be much darker than the actual color.

Case C shows a ray which strikes the bald ground underneath the tree. As previously stated the database only holds a single gray shade value for this position in the database. Where the tree is surrounded by grass this is not a problem, but this is seldom the case. This is a short coming of the database which cannot be readily overcome. Presently, the function returns the gray shade value of the tree. One approach to resolving this problem would be to code the function to "remember" the last value of the gray shade where the 'uci' was zero (i.e. bald terrain). When this particular case was encountered, the last bald terrain value could be returned, vice the gray shade value of the tree.

Case D is the simplest one to handle, with a 'uci' equal to zero, the gray shade value is returned without further consideration.

Case E requires special handling, with many possible solutions. Case E is a ray which passes beneath an overhang, but does not strike the object. The gray shaded grid positions beneath the tree indicates the positions where the 'uci' is greater than zero, but the ray is below the level of the 'uci'. In this case, no gray shade value should be returned and the ray trace should be continued. The options are to restart the ray trace recursively, or to set a flag and allow the ray trace to continue until the flag is cleared. This last option is the one implemented in the improved function HITGRNDA.

As can be seen from the flow chart and the code in the appendices, the cases are handled in the order presented above using chained 'if' statements. This version of HITGRND also incorporates the grid line code. The 'col' variable is used as the flag to allow the ray trace to continue. Gray shade values range from 0 to 255. Setting 'col' to 300 effectively sets a flag and causes the ray trace loop to continue executing. This implementation requires modification to the ZIGZAG function in order to execute properly. The modified function appears in the appendices as ZIGZAGA.

G. GETTING THE IMAGE TO THE SCREEN

Once the VIEW Vector has been filled with gray shade values the scene can be written to the screen. This is accomplished by the function DISPLAY. This function is essentially comprised of two graphics commands. The first is 'rectzoom', which scales the source image in the 'x' and 'y' directions by a vector of scale factors passed to the function. For the PVG the image is scaled by the same factor in both directions. The scale factor used is the one input during program initialization. 'Rectzoom' does this by taking each pixel in sequence

from the source image and expands it in to multiple pixels by the scale factor[Ref. 4]. Figure 6 shows how 'rectzoom' expands pixels prior to writing to the screen.

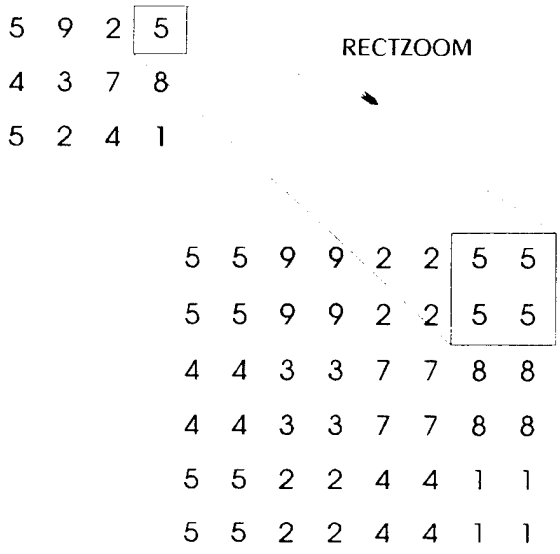


Figure 6 Pixel Expansion Using 'rectzoom'.

After Ref. [4].

Once the image has been scaled for the window it can be written, one pixel at a time, to the screen using the command 'lrectwrite' (Figure 7). The Cartesian coordinates for the lower left and upper right hand corners of the pixel grid must be specified. This provides the function with the height and width of the window, or portion of a window being written to. By default, pixel writing begins at the lower left hand corner, from left to right, and bottom to top. The order that pixels are written can be selected using the 'pixmode' command[Ref. 4].

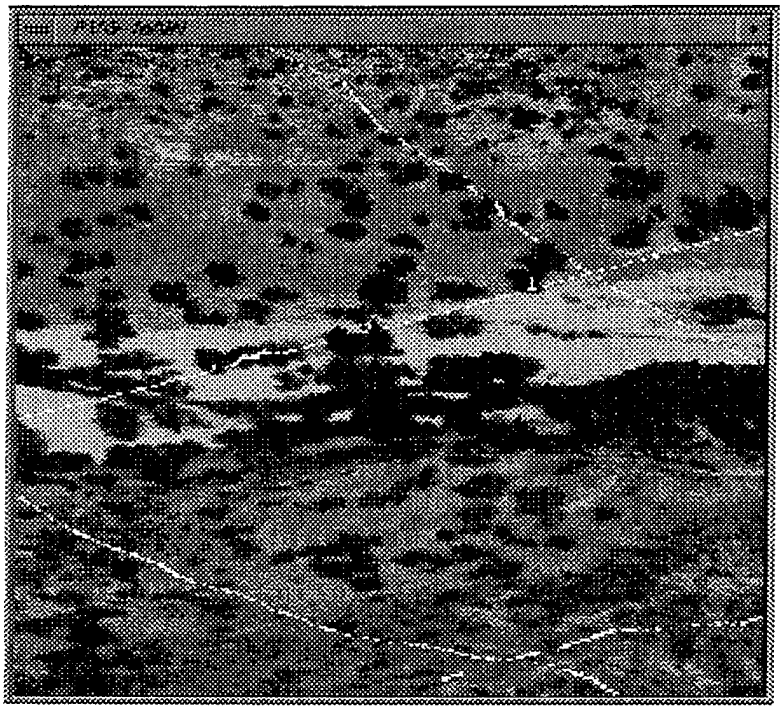


Figure 7 Sample PVG Scene.

III. OPTIMIZING THE SERIAL PROGRAM

A. TEST PROCEDURES

The major issue to be resolved with respect to the PVG is the "speed" of the algorithm. In the context of a computer program, "speed" can take many forms. First, there is the amount of time the processor spends executing the program, or CPU time. Next, there is the time input/output (I/O) devices take to process and display information. This can be a considerable challenge to measure due to the interaction of memory buffers with the CPU and the I/O device. A related question is, "What are we measuring?" Are we interested in the total execution time of the program, or the amount of time a particular loop takes to execute? Relative time is another issue, "What portion of processing time is occupied by one operation versus another?"

There are several approaches to measuring the speed of a program. They range from well constructed and precise tools contained in the operating system to more traditional methods involving stop watches and statistics.

Code profiling. This is accomplished using a UNIX feature, 'prof'. To set the program up for profiling a switch, '-p' , must be included in the command to compile the program. This switch compiles the program with breaks every 10 milliseconds to sample the function the CPU is presently executing. If one is using the total execution time of the program to compute another value, the overhead of collecting the data on the currently executing function introduces an uncertainty which is extremely difficult to estimate. Log810 (Appendix C) shows an example of the output produced by the 'prof' command. From this

example it is clear that the relative time spent on each function is easily obtained, as well as total execution time.[Ref. 4]

Pixie Profiling. This tool is very similar to the 'prof' command. However, 'pixie' and 'pixiestats' provide a finer degree of granularity and a wider range of statistics. The price the programmer pays for these advantages is the greatly enlarged overhead of the electronic book keeping. Referring back to Barr E. Bauer's strategy for parallelizing code, 'pixie' and 'pixie stats' can be used very effectively in accomplishing the first step of this strategy. 'Pixie' and 'pixiestat' also have the power to analyze the activity of individual threads of multi-processor code[Ref. 4]. This is of particular value in resolving load imbalances between threads in multi-processor code. "Threads" and 'Load Imbalances" will be discussed in greater detail later in the thesis.

Log98a (Appendix C) shows several examples of pixie output files. In addition to the much finer granularity, a number of additional measured values are available. For the purpose of analyzing and improving speed, the data on instruction cycles lost or delayed (contained in the file header) can be very useful. From this data the effect of adding and deleting input and output devices can be assessed. An inspection of the basic-block counts can also give the programmer a feel for the interaction of operating system features with the program. In the first example in log98a the function using the most cpu cycles is "_mp_slave_wait_for_work". This function is part of the multi-processor scheduler, not formally a part of pvg-main.c.

Manual Timing of programs. For the PVG the only portion of the program of interest, as far as speed of execution is concerned, is the 'do while' in pvg-main. Each pass through this loop represents one frame. Referring once again to log810, two trials are shown. The

first trial shows the total execution time of the program to produce 40 frames. The times were obtained using the 'prof' function. One might be tempted to divide 40 frames by 5.6 seconds to obtain a frame rate of 7.1 frames per second, including the time spent loading the database. This frame rate would indicate a video rate which is approaching continuous motion. What the user of the PVG actually observes is about one frame per second, far from continuous motion. A less precise, but more accurate method is to time the period between two events on the screen. This method was employed in the second set of trials. These times were obtained by starting a stop watch when the first frame appeared, observing the screen move 39 times and stopping the stop watch. By this method, the frame rate was determined to be approximately 1.1 seconds. For the purposes of determining frame rates manual timing methods will be relied upon and use of the 'prof' function will be limited to making comparisons between programs.

For rates of four frames per second and less, counting the frames is acceptable. As the frame rates rise from this level it becomes increasingly difficult and eventually impossible to obtain accurate frame rates. It was inevitable that the PVG would have to be modified in order to facilitate manual timing. The solution to the problem can be seen PVG-noV. The 'do while' loop has been modified slightly to count 60 frames and terminate. Two print statements were added to signal the start and end of the frame generation portion of the program. On obtaining the desired frame rate of twenty frames per second the interval between the start and end messages would be approximately three seconds. This is an interval which can be easily measured with a stop watch. Appendix D contains the specific manual timing procedures used to obtain frame rates.

Manual timing introduces a number of uncertainties into the testing process. The first is the uncertainty in the timing device. While on the order of hundredths of a second for an electronic stop watch, this is still high when compared with clock error in the Silicon graphics machines. Second is the inherent uncertainty of a human operator. Another consideration is that the machines available for testing are predominantly servers. In all cases they are 'time sharing machines'. The seconds lost to time sharing and servers is not included in the 'prof' and 'pixie' output. The number of people who are sharing a processor varies over the course of the day, but over the span of testing can be taken to be random. The effects of sharing the CPU can also be reduced by testing during periods of low CPU activity. All this leads to the conclusion that statistical analysis and trials of greater than 30 measurements are required to obtain statistically valid results [Ref 5].

The following table provides a summary of the machines used in testing the various versions of the PVG discussed in this thesis. Algieba, Alioth and Tobago are servers for specialized programs for their respective departments. Indy2 is a single processor work station which is typical of the machines used to perform serial optimization.

Machine Name	CPU	Clock Speed	Data Cache Size	Instruction Cache Size	Main Memory Size
Indy2	R4400	100 MHZ	16 Kbytes	16 Kbytes	32 Mbytes
Algieba	2xR8000	75 MHZ	16 Kbytes	16 Kbytes	384 Mbytes
Alioth	8xR8000	33 MHZ	64 Kbytes	64 Kbytes	256 Mbytes
Tobago	4xR4400	150 MHZ	16 Kbytes	16 Kbytes	128 Mbytes

Table 2. Summary of Machines Used In Development and Testing

B. MODIFICATIONS WHICH HAD NO EFFECT

Before proceeding with Bauer's strategy for parallelizing an algorithm, the serial program should be optimized to the greatest extent possible. Considerable effort had previously been expended by Baer and Driels in this area. This limited the opportunities for further optimization. The first area which was investigated to find further optimizations was in the selection of loop structures and syntax. Each loop structure; 'for', 'while' and 'do while' each incur their own overhead in the executable code. Various combinations of loop structure were employed within ZIGZAG and the frame rates recorded for each. Log810 and log823 in Appendix D show the raw results from this testing. None of the three loop combinations varied by more than one standard deviation from the baseline version.

C. THE HORIZONTAL ZIGZAG

One version of ZIGZAG tested did actually improve performance by more than one standard deviation. This version employed a horizontal ZIGZAG similar to the vertical terrain following zigzag. An additional variable was included to hold the value of 'step' at first terrain contact from the previous column at row zero. The baseline program took 38.45 seconds to generate 40 frames on average with a standard deviation of 3.74 seconds. The horizontal zigzag version required 31.84 seconds with a standard deviation of 2.64 seconds. As the time difference between the two versions is greater than their combined standard deviations, the improvement can be considered statistically significant.

This implementation may at first seem appealing, but has a serious drawback. Objects which are tall and have a base closer to the observer than the first terrain contact will be

passed over (see Figure 8). Ray trace 'a' is the first ray trace that ZIGZAG would perform. With the horizontal ZIGZAG the ray trace would start each successive ray trace at a point three steps back from the dotted line connecting 'a' and 'b'. Since this line passed behind the telephone pole in the fore ground, the PVG would never "see" the object.

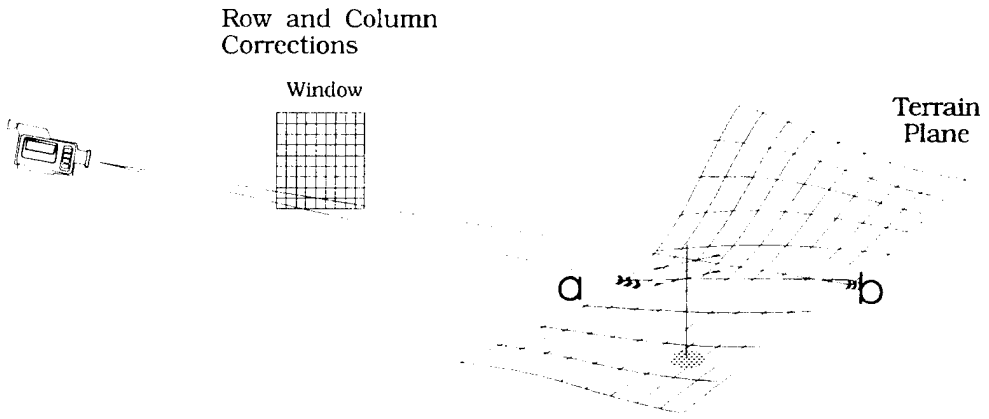


Figure 8 Horizontal Terrain Following.

IV. PARALLELIZING THE PERSPECTIVE VIEW GENERATOR

A. AN INTRODUCTION TO PARALLEL PROGRAMING

As the demand for speed in information processing increases, and the opportunities for technical breakthroughs which would improve processor speed become less likely, the pressure is mounting to find alternatives to serial processing. Multi-processor architectures offer a promising alternative which could improve speed by a factor equal to the number of processors available. The multi processing architecture by itself will not produce improvements in performance. Algorithms must also be designed to run in parallel. The appropriate code must then me inserted to signal the operating system that portions of the program can be run simultaneously.

Parallel processing algorithms fall into three categories.

Result Parallelism - Focuses on parallel computation of all elements in data a structure[Ref. 3: p. 11].

Agenda Parallelism - Specifies an agenda of tasks for parallel computation of distributed data that is accessible by many processes simultaneously[Ref. 3: p. 11].

Specialist Parallelism - Specialist agents solve problems cooperatively in which data is passed between the agents[Ref. 3: p. 11].

Simple algorithms, such as performing a character string search of a very large data base, would fall neatly in to the category, Result Parallelism. Once algorithms move to a higher level of complexity it is unlikely that they would fit so neatly into a single category. For the PVG, the ZIGZAG function taken by itself, appears to be an obvious case of Result

Parallelism. But when considered in relationship to GENDCOS and DISPLAY does it become Specialist Parallelism. Especially if the DISPLAY function can also be parallelized.

One might wonder what the point of categorizing is, if an algorithm can so easily fall in to multiple categories. Power C provides a wide range of tools and syntax elements which can be used to parallelize an algorithm. Different tools in the Power C tool box are more appropriate to some categories of parallelism than others. Some tools can be adapted to for use on any type of loop, or to run heterogeneous fragments of code in parallel. Other syntactical elements are only applicable to 'for' loop structures. By categorizing an algorithm the programmer can make some preliminary decisions about which tools should be investigated.

An important concept that is fundamental to multi-processing is the idea of a "thread". A thread is any fragment of code which can be executed simultaneously with other sections of code. The condition which prevents a code fragment from being run in parallel is data dependence. Specifically, two fragments of code may need to modify or read the same data element simultaneously. Syntactical elements exist in Power C to deal with these conditions and allow a thread to remain independent. In other cases the code may have to be modified to eliminate the data dependence so the thread can run in parallel with other code.

One example of data dependence is the case where a value is carried from one iteration of a loop to the next, and modified. The code fragment below shows a simple

```
cvar = 0.0
do i=1,1000
     a(i) = cvar
     cvar = b(i) / c(i)
enddo
```

example of this.[Ref. 3: p. 50] The value of 'cvar' is carried from forward from the previous iteration and then modified. The current value of 'cvar' is dependent on the results of the previous iteration of the loop.

Another fundamental concept of parallel programing is the concept of "load imbalance". Ideally, if a serial program is modified to run on a four processor machine, there should be a four fold increase in speed. Some small overhead must be paid in initializing the parallel region, but only a small portion of one CPU's time will be occupied by the operating system scheduling CPUs. Where the ideal is not realized, the most likely cause is load imbalance. This occurs when one thread takes significantly longer to execute than other threads running in parallel. In the four processor example, instead of executing in just over one quarter of the time, the program will execute in the time is takes to execute the longest thread. In a poorly designed algorithm this can be significant.

Log817, in Appendix C, provides an example of load imbalance in action. This log was generated to demonstrate proper parallel operation of Alioth after a software patch was installed. The text of the program appears in Appendix B as 'testfn5g.c'. This simple program is designed to print 'block 1',' block 2',' block 3',' block 4' in order with a line feed after each. By adjusting the load in each loop within the independent blocks, the order of printing is changed. This is accomplished by altering the values for the variables 'alt1','alt2','alt3' and 'alt4'.

B. THE KEY #pragma LEXICON ELEMENTS, WITH EXAMPLES

The Power C library provides both the syntax to allow a programmer to manually parallel an algorithm, or a routine to generate parallel code. Rather than cataloging the commands here and explaining their purpose, the few syntactical elements which are relevant to parallelizing the PVG will be reviewed here.

#pragma parallel - This compiler directive is used to define the region where parallel code resides within the program. Multi-processor directives outside this region are ignored. The parallel region is most often bounded by '{' immediately following the #pragma and '}' at the end of the parallel region[Ref. 3: p. 145]. Where the parallel region is limited to a code fragment bounded by another '#pragma' immediately following the '#pragma parallel' the brackets can be omitted.

#pragma independent - This is the most conceptually simple syntax which can be used to define a parallel thread. The executable fragment is enclosed in brackets as in the above '#pragma'. The programmer assumes responsibility for ensuring that there are no data dependencies which would cause segmentation errors or erroneaous answers[Ref. 3: p. 145-146]. Any code contained between the brackets of an independent block can be executed in parallel with any of the other code in a parallel region. A sample of the syntax for this '#pragma' can be found in Appendix B PVG-MAIN2.C and testfn5g.c.

#pragma pfor - This element supplements the 'for' command in a program. The '#pragma' applies to the 'for' loop immediately following it. The iterations of the 'pfor' loop are split amoung the available processors at compile time.[Ref. 3: p. 145] Once again, the programmer is responsible for ensuring that no value computed in the a loop is required in a

succesive loop, such as accumulating a sum. An sample of the 'pfor' syntax appears in the text box below. The program assigns zero to each element of 'piglet', an array of 1000x1000 elements.

```
void main (void)
{
     int i,j,k;
     int piglet[1000][1000];

     i=0;
     while ( i<1000 ) {
#pragma parallel shared(k) local(j)
#pragma pfor iterate ( j=0;1000;1)
     for ( j=0; j<=1000; j++) {
          piglet[j][k] = 0;
     printf("Complete through cycle number %2d.\n",i);
     i++;
    }
}
}
```

The above code is actually an example of a nested loop. In the line containing the '#pragma parallel' two code elements signal the compiler which indices will be split among the available processors, and which will be duplicated in each thread. The 'local' variable 'j' will be split. In this example, if two processors were available, one would take 0 to 499 and 500 to 1000 would go to the second processor. The inner loop will be duplicated in each of the threads, hence the 'shared' variable. The compiler cannot actually allow one variable of the same name assigned two memory addresses (one for each parallel thread) so it creates two dummy variable to take the place of the 'k' in each loop. This is all transparent to the programmer as these dummy variables can only be found in the code just prior to linking.

C. METHODS OF PARALLELIZING CODE

For very simple algorithms the Power C Analyzer (PCA) can provide a "quick and dirty" method of producing parallel code. The PCA will take an input file written in ANSI C, perform a serial optimization of the code and identify elements which may be parallelized. On identifying parallelizable structures, it converts 'for' loops to 'pfor' loops. The PCA is unable to distinguish any other threads which can be placed in parallel regions,[Ref. 3: p. 382-386] although it will allow manually parallelized code to pass through unmodified. Mixing manually parallelized code with the PCA generated code the programmer may inadvertently violate Bauer's strategy by parallelizing small loops and loops with short execution times.

In employing the PCA Bauer provides some guidelines.

- Transform 'while' and 'do while' loops to 'for' loops.
- Focus on loops with finite limits and integer increments.[Ref. 3: p. 304]

A careful examination of the three loops of the ZIGZAG functions shows why, at first, the PCA appears to be an appropriate tool for this problem. The 'do while' loops cannot be converted to 'for' loops since they do not use an integer index, but one loop in the ZIGZAG function has a true integer increment. Fortunately, this is the outermost loop. If one were to parallelize this loop within the function this would be the place to do it. In practice the PCA will not convert this for loop. By default, the PCA will not convert any 'for' loop where the PCA estimates the loop contains less than 1000 executable instructions. The PCA is unable to estimate the number of executable instructions in a 'do while' loop, resulting in a count of zero. This default value can be set to zero, which should allow the

conversion to proceed. Once this hurdle is cleared the PCA then encounters another obstacle to converting the 'for' loop. Due to the use of the indices from the two outer loops to access elements in the array DAT the PCA sees variable dependencies between succesive iterations of the of the 'for' loop where there are none.

The PCA can be invoked as a stand alone command or by including '-pca' as a switch when using the 'cc' command in UNIX. When manually generating the code the switch '-mp' must be included to call the multi-processor library and tell the compiler to set up multi-processor compilation.

D. THE PARALLEL VERSIONS OF PVG-MAIN

From the previous discussion it is clear that an automated parallelization of the PVG program was problematic. The PCA provides enough switches and compile parameters that executable code could have eventually been produced. It is not clear that this would have actually improved the code. The PCA would also have had to be invoked from within the makefile. The interaction of the PCA's serial optimizers and the -O2 and -O3 optimizers available in the UNIX operating system is not well documented. For these reasons a manual implementation was pursued.

The level in the program where the code would permit parallelization was dictated by the nature of the ray tracing algorithm. Prior testing by Baer[Ref. 2] and Driels[Ref. 1] demonstrated that a majority of the time spent by a serial processor was concentrated in the ZIGZAG function. Within the innermost loop of ZIGZAG each iterative step depends on the results of the previous iteration. These types of loops are virtually impossible to

segregate. Parallelizing the inner most loop also leaves a larger portion of the code outside of the parallelized region.

The next higher loop advances the ray trace row by row. This is also the loop where the terrain following feature is implemented. If this loop were partitioned, the ray trace would be restarted once per column for every additional processor. This would reduce the advantage gained by implementing the terrain following feature. Once again, a portion of the code would be left outside the parallel region which could otherwise be included. The natural partition point is the outer most 'for' loop.

Partitioning the 'for' loop can be accomplished by one of two ways. The most direct approach would be to implement a '#pragma pfor' within ZIGZAG. The second option is less direct, but equally effective. The ray tracing can be partitioned by columns from outside the ZIGZAG function by specifying the beginning and end column numbers in PVG-MAIN.C. The required changes are as as indicated in the flow charts in Appendix A for PVG-MAIN2.C, PVG-MAIN4.C and PVG-MAIN8.C. The corresponding code is contained in Appendix B. The numbers in parenthesis next to the function call indicate the column partitions. This implementation requires no modification of the ZIGZAG function.
The indirect method was selected because it gives the programmer a higher degree of control over how the work is apportioned. Additionally, it is conceptually simpler, with clean and understandable code.

E. SUMMARY OF FRAME RATES

The table below presents the frame rates for all versions of the PVG tested on all the available machines. The effects of employing multi-processing as well as new technology are evident. Each machine should have experienced a speed increase nearly equal to the number of processors it employs. The machine that came the closest to attaining this standard was Algieba, which attained 82% of the expected increase. The machine that did the poorest was Alioth, attaining 55% of the expected increase. Notably, Algieba is the newest machine, and Alioth represents technology which is now considered obsolete.

Machine	Speed Increase Serial/ Parallel with Video	**Test Version**				
		PVG Serial No Video	PVG Serial With Video and Mouse	PVG No Video, Multiple Proc.	PVG Video, No Mouse, Multiple Proc.	PVG, Multiple Proc. With Video and Mouse
Algieba	1.62	6.6	3.6	12.4	5.6	5.8
Tobago	2.68		3.4	10.8	9.5	9.1
Alioth	4.45		1.11	6.3	3.3	4.9

Table 3. Frame Rates for Versions of PVG Tested and Machine Tested On

On completing the testing on Alioth the Pixie profiling tool was used to profile the execution of the eight thread program. The results are contained in log98a in Appendix C. The first two profiles are both versions of the PVG with video, the second profile is of the version without a mouse. Comparing the two the effect of adding the mouse is indirectly evident. The portion of time spent by the processor doing productive work , 43.79% for ZIGZAG, increased from 34.45 %. The picture for this particular machine becomes even

more muddled when it is observed that the frame rate increased after adding the mouse. Tobago showed a more predictable result with frame rates decreasing by approximately 4%.

Considering that Algieba attained 96% of the theoretical maximum, without video, it appears that the ZIGZAG algorithm is at or near the best performance which can be achieved using multi-processing. It was expected that adding video would reduce the frame rate as well the percent of the theoretical maximum. This is a direct consequence of adding code to the program outside of the parallel region.

By observing the change in frame rates, after adding video, we can estimate the impact of adding video to the PVG. In all cases there is a significant drop in frame rates (37.9% Algieba, Tobago 15.6% and Alioth 21.5%). By examining the Pixie profiling in log98a we can gain further insight into the impact of adding video. After video is added only .37% of the CPU time is spent executing functions explicitly identified as GL functions. Considering that GL functions may be calling other functions, the time spent by the CPUs working on graphics related functions does not appear to be more than 4.5%.

Comparing the profile for pvg-noV8 with pvg-V8 one major difference is the percent of time CPU cycles occupied by '_mp_slave_wait_for_work'. This goes from 3.42% without video to 48.59% with video. There are at least two possible explanations. The problem could be hardware related, the assigned CPUs are waiting for I/O devices to release the buffer. Another possibility is that the other seven processors are left waiting for long periods of time for the single processor working on the graphics to finish. If the problem is hardware related it is expected that the drop in frame rates for the newest machine (Algieba) will be by a smaller percentage than on the older machines.

F. TOPICS FOR FURTHER RESEARCH

It appears the loss of performance is due to the DISPLAY function being in the parallel region. Therefore, more code from the DISPLAY function will have to be moved to the parallel region to improve efficiency. As a test to see if the 'lrectwrite' statement could be parallelized the test function in the VTEST.C (Appendix B) was run. The function consistently crashed the video server on both Tobago and Alioth. Under the current limitations of the architecture the screen write cannot be parallelized.

In spite of the tests cited above, this line of research may not be hopeless. Due to the problems with running GL graphics on Algieba the VTEST.C program could not be run on that machine. As Algieba represents the best technology available from Silicon Graphics, Inc. upgrades in the hardware and software in this series which could allow multiple processors to write to the screen are not out of the question. The possibility that OpenGL may permit these type of screen writes has still not been fully explored.

Another option for parallelizing the graphics process would be to replicate the 'rectzoom' command in parallelizable code. This would require the VIEW vector to be increased by the scale factor squared. The function INIT_WINDOW would also have to be modified to provide a larger window of the appropriate size. The equivalent code which performs the 'rectzoom' function would then be embedded in the ZIGZAG function. For example, if the scale factor were 2, a single pass through the ray tracing loop in ZIGZAG would cause four values to be written to the VIEW vector rather than one. Each ZIGZAG function would perform the 'rectzoom' for its own column partition, effectively splitting the 'rectzoom' among the available processors.

V. REPRESENTING TARGETS IN THE PERSPECTIVE VIEW GENERATOR

A. PRELIMINARY EFFORTS TO INTRODUCE TARGETS

In Baer's SOPVG targets were represented using video objects known as "sprites". Sprites come in sets. Each set of sprites depicts a single vehicle, or man made object from a series of equally spaced aspects. A list of the positions and identities of all man made objects moving through the terrain represented by the database are maintained by the SOPVG. Once the scene is generated, at each position of a man made object, a sprite, with the closest matching aspect to the observer is selected. The sprite is scaled to the appropriate size and electronically pasted into the image.

This method takes the best elements of the ray tracing algorithm and the necessary elements of an object oriented approach to render a scene. Although the sprites very accurately portray man made objects, they can only portray a limited number of aspects. As frame rates increase and approach continuous motion for the terrain, the sprite only changes when the boundary from one aspect to the next is crossed by the observer. The effective frame rate of the sprite is actually less than that of the surrounding terrain.

In implementing targets in the PVG for use in the JANUS(A) gaming system the programmer must be concerned with the target's appearance being consistent with the background. One of the objectives of the gaming system is for the user to identify and classify targets in the same manner they would in the field. Where one portion of the of the scene behaves in a manner that is not consistent with it's nature, it can give the user an advantage they may not have in the field. Therefore it is highly desirable to find a means of rendering

man made objects in the system that not only accurately depict them, but are also consistent with the nature of their movement.

In Driels earlier work, a very simple vehicle measuring 4 meters by 4 meters by 10 meters was employed to explore the fundamental geometry of acquiring targets in the PVG. The target was applied to the database by adding 4 meters to the value, 'ele', at each grid position of the vehicle foot print. The function TARGET accomplished this, as well as advancing the vehicle in a hard coded direction of motion. This resulted in a vehicle which is conformal rather than rigid. Figure 9 illustrates how this algorithm causes the vehicle to assume the contour of the terrain. Due to the oblong shape of the vehicle it could only move

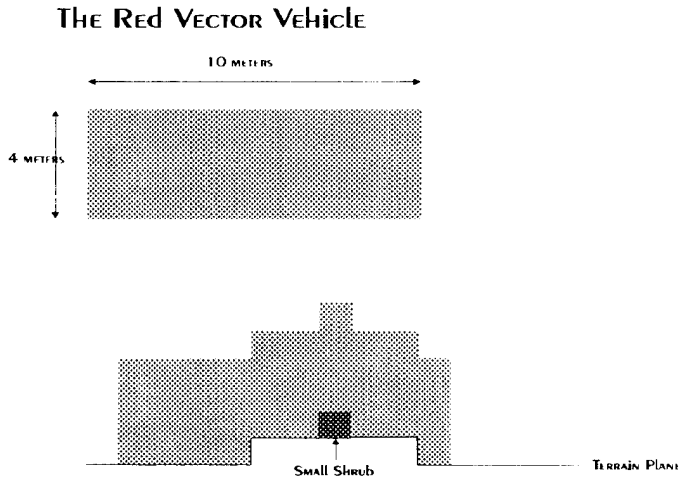


Figure 9 Driels Original Test Vehicle.

along the axis of the Base Elevation Plane without further distorting the geometry of the vehicle. In some of the more extreme cases the target could be observed to be crawling over the tops of trees, more like a caterpillar than a man made vehicle. The function RESTORE reset the value of 'ele' to it's original value.

The next level of complexity to be explored was the representation of an actual vehicle at one meter resolution. The first step in this process was to obtain a line drawing of a Russian T-72 tank. This drawing was superimposed on a grid where each block measured one meter by one meter. For the most part, any block more than 50% occupied by the profile of the tank was filled with a color. Artist's license was taken where appropriate in attempting to make the block resemble a tank. The resulting target, side and top view, is shown in Figure 10.

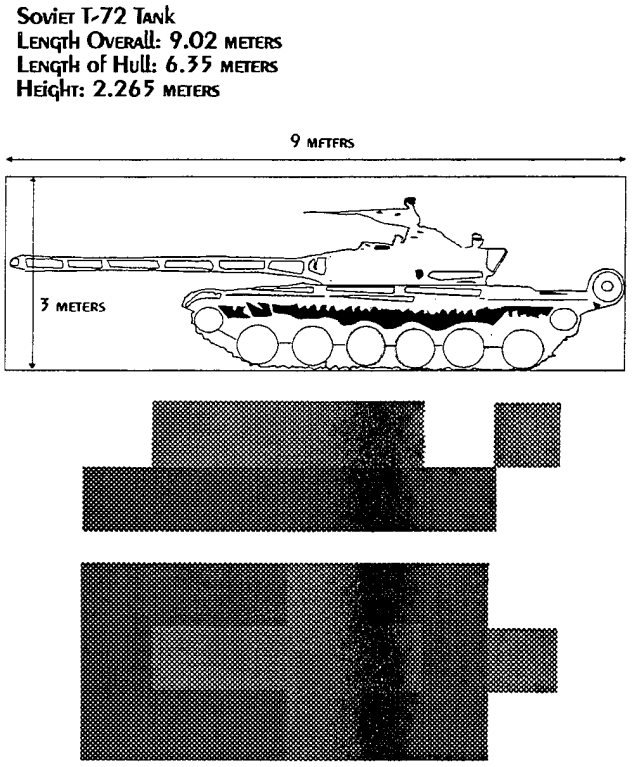
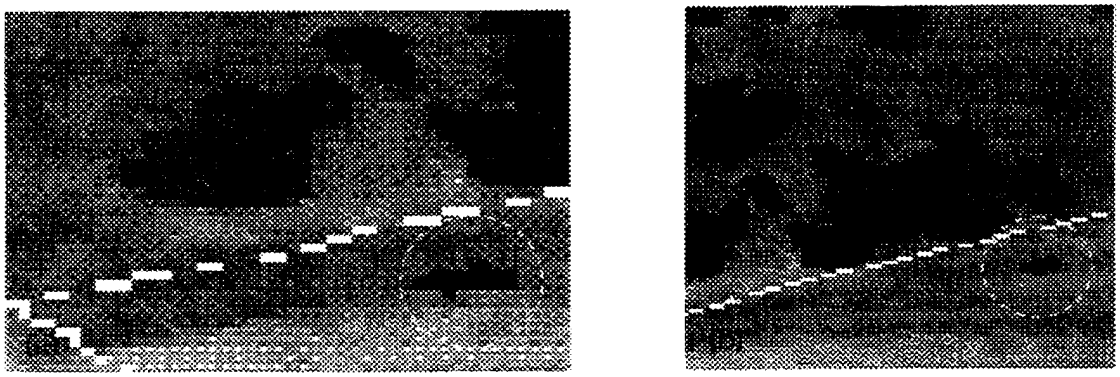


Figure 10 One Meter Resolution Target. After Ref.(6)

This one meter resolution target was implemented in a manner similar to the red vector vehicle. Additionally, it was maneuvered parallel to an axis of the Base Elevation Plane in clear, relatively flat, terrain. The purpose of this experiment was simply to get a rough idea of whether the target would be recognizable as a tank. Figures 11a and 11b show two snapshots of the vehicle moving in open terrain. The one meter resolution vehicle appears in the white dashed circle. The distortion due to the conformal implementation is particularly evident in Figure 11a.



(a) (b)

Figure 11 Screen Capture of One Meter Resolution Target.

In addition to the disadvantages of the red vector vehicle already mentioned several more issues were brought to light observing the one meter resolution tank moving over the "level" plane. The conformal implementations fail to consider the angle of the grade when climbing a hill or tilt of a vehicle as it runs perpendicular to the slop. In Appendix B the text of the function of TARGETA shows how cumbersome the code can be for directly applying the features of the one meter resolution target to the database. Of course everything which

is done to the database must be undone. The function RESTOREA, which clears the effects of TARGETA is only slightly less cumbersome.

B. TOPICS FOR FURTHER RESEARCH: A MORE REALISTIC TARGET

The first, and simplest problem to be tackled was to render a higher resolution target for the database. This was accomplished using the method outlined above for the one meter database, substituting a .25 meter grid for the one meter grid. This provided the much better representation found in Figure 12. The question now arises, "How do you represent an object with quarter meter resolution in a one meter resolution data base?'

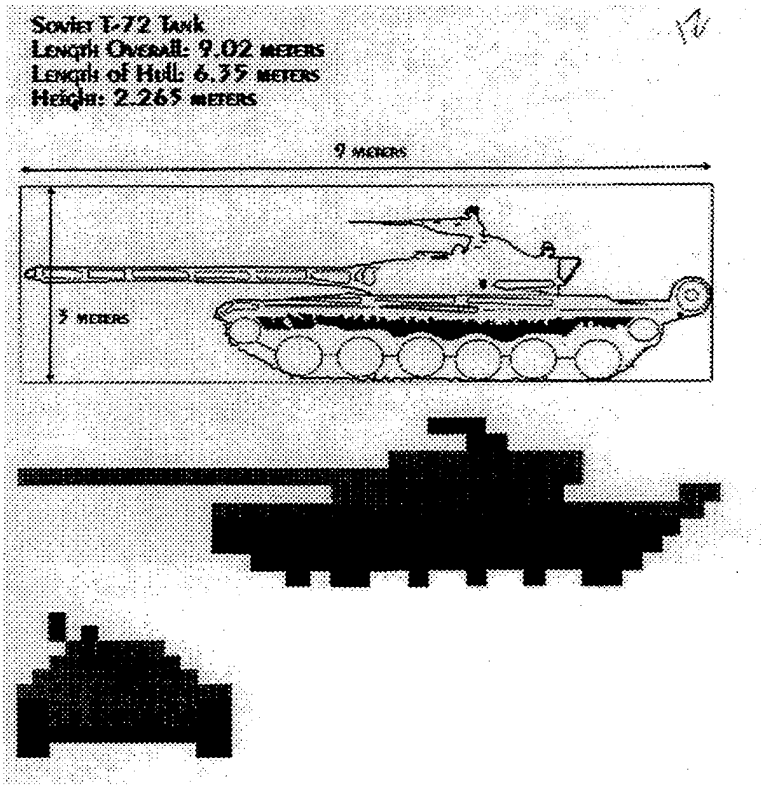


Figure 12 A One Quarter Meter Resolution Target. After Ref.(6).

One recommended approach is to establish additional plane of reference with another set of transformations for a mini-vehicle database. This database would be fully three dimensional and the coordinate system would be relative to the vehicle. This vehicle database would be represented in the greater one meter database as a conformal symmetric box. When a ray trace reaches one of these symmetric boxes the ray is rotated in to the vehicle plane of reference. The ray trace would then be continued in one quarter meter steps until the vehicle is contacted. As in the HITGRND function, a color is returned once the contacted surface is resolved.

As the boxes are symmetric they can easily be maneuvered on the Base Elevation Plane without distorting the box. Figure 13 shows how the symmetric box would be sized.

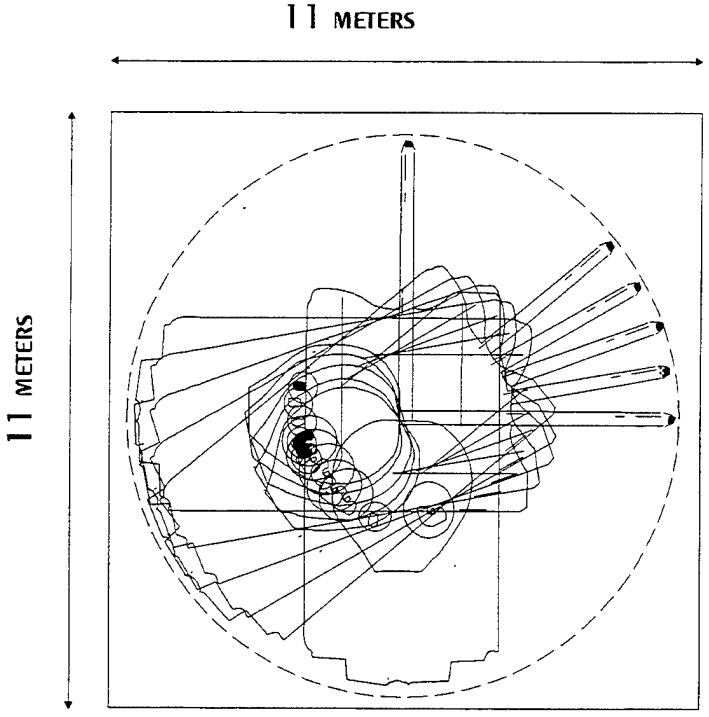


Figure 13 Sweep of Vehicle Inside Symmetric Box.

The geometric midpoint of the vehicle axis would be found. The vehicle would be rotated around this point and the sweep of the vehicle determined as if it could pivot on that point. The box would then be sized such that it was a whole meter value on each side and contained the entire sweep of the vehicle.

Figure 13 also helps illustrate one of the transformations which would have to take place. The true heading of the vehicle would be compared to the true bearing of the observer to obtain the relative bearing. This would be the angle off the axis of the vehicle where the ray trace would begin.

Figure 14 shows how the height of the symmetric box would be sized in order to allow the conformal feature of the implementation of the symmetric box to work. It is

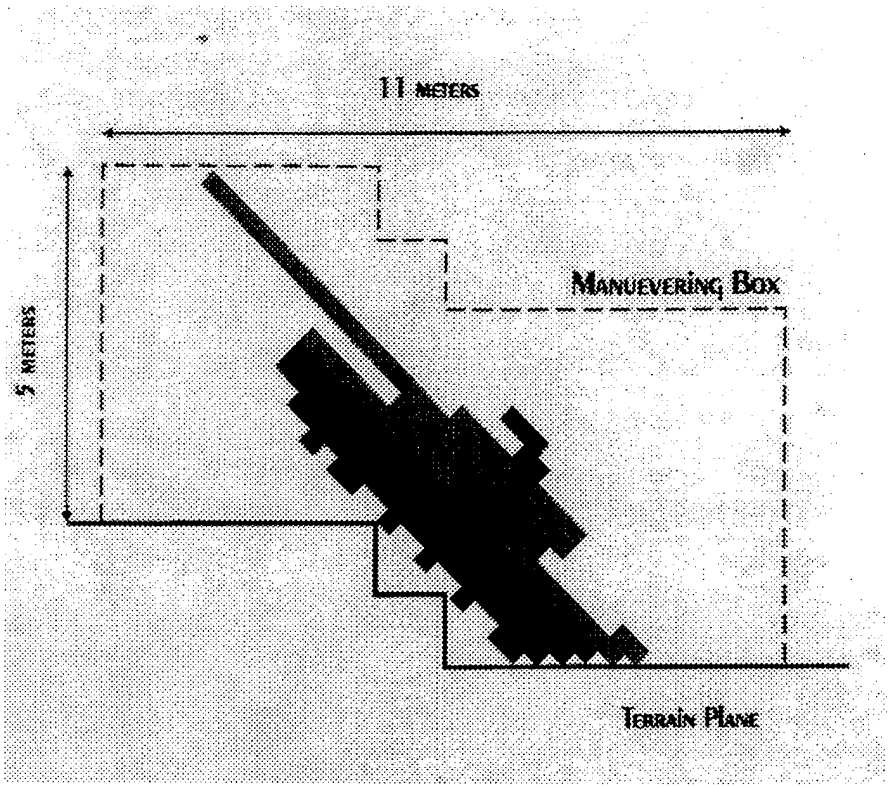


Figure 14 The Maneuvering Box in Profile.

conceded that these boxes may be more empty space than vehicle, and that these empty spaces will slow down ray traces which would otherwise contact bald terrain. Further research will be required to assess how the movement of these boxes will be affected by terrain, overhangs, vegetation and other vehicles.

Figure 15 illustrates how various ray traces cases would be rotated in to the vehicle database. In case A it is a simple trigonometry problem to compute the point where the ray intersects the base plane once the angle of climb and the point where the ray intersects the box is known. Similar calculations would be conducted for the perpendicular case: tilt. Case A would clearly intercept the upper surface of the tank, assuming the ray trace is near the

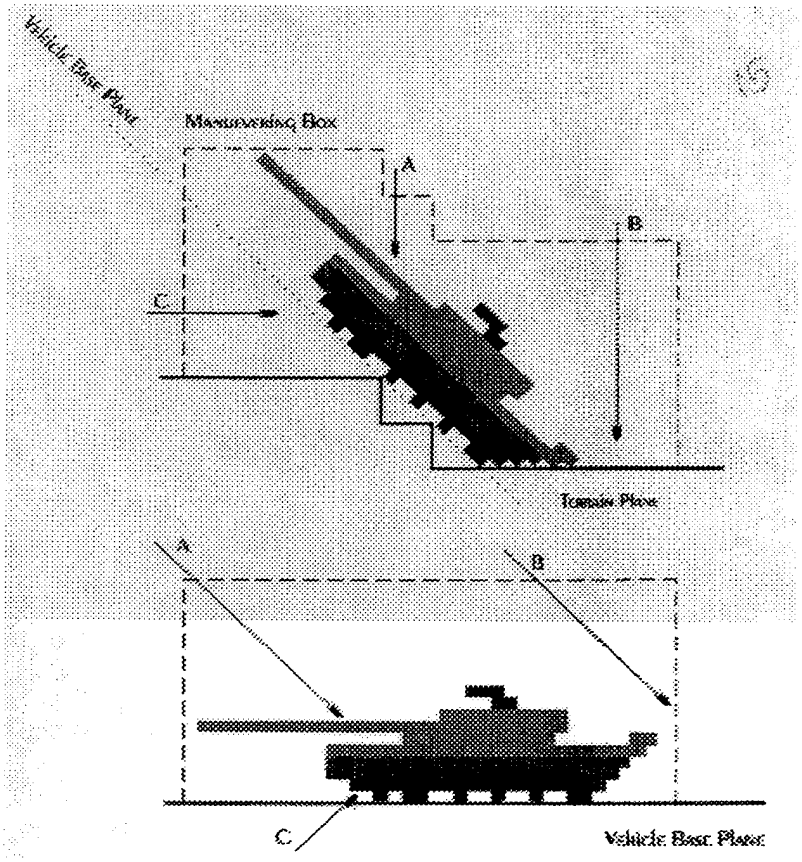


Figure 15 Ray Traces Inside the Symmetric Box.

centerline.

Case B shows how a ray could enter the box and not contact the tank. In this case, when the ray reached the limits of the box and was about to leave, the ray would have to be transformed back to the PVG database's reference plane.

Case C shows one of the strengths of this approach. On hilly terrain or around ravines the bottoms of vehicles are often exposed. The strictly conformal approach is incapable of rendering this view. Sprites simply do not include this aspect.

The database used to define the T-72 tank from Figure 12 would consist of 2420 thirty-two bit words. The position in the data base would correspond to a .25 meter square cube in a three dimensional array. The 32 bit word would be used to describe the qualities of the part of the vehicle which is contained in that cobe. The vehicle database, unlike the terrain database, would be fully three dimensional. The fully three dimensional structure provides much more flexibility in how the 32 bit word can be used. It could be used to store up to 4 colors, or three colors and a reflectivity index.

Only one vehicle database would have to be maintained for each type of vehicle. For vehicles with turrets the opportunity also exist to split the database to permit rotating turrets. An additional table would have to maintained holding an identification number for all vehicles on the terrain plane, their position and other vital statistics. These statistics would include true heading, angle of climb, tilt and turret bore sight angle. Algorithms will have to be developed to analyze the terrain covered by the foot print and determine if the vehicle can navigate the terrain. Once it has been determined that the vehicle can operate on the terrain the tilt and angle of climb must be determined.

VI. CONCLUSIONS

It is clear that the database oriented virtual environments coupled with ray tracing algorithms represent a viable means of rendering accurate terrain scenes. The target speed of 20 frames per second should be easily achievable without technological breakthroughs. Employing the same processors as used in Algieba, in an eight CPU architecture would very likely reach the 20 frame per second goal. This particular processor is available from Silicon Graphics in architectures employing up to 32 CPUs.

When implemented using multi-processing architectures, the potential exists for greatly improving resolution. The higher speeds which are achievable can be traded off for higher resolution databases. The surplus speed would also permit more sophisticated rendering of targets, as well as terrain and vegetation.

The potential for developing high quality virtual environments in parallel architectures has only barely been tapped.

LIST OF REFERENCES

1. Morris Driels and Judith Lind, "Prototype Line of Sight and Target Acquisition Software for JANUS(A) High Resolution Terrain Databases," Final Report FY95.

2. Wolfgang Baer, "Implementation of a Perspective View Generation Replicator," Nascent Systems Development, Carmel Valley, California.

3. Barr E. Bauer, Practical Parallel Programming, Academic Press Inc., San Diego, California, 1992.

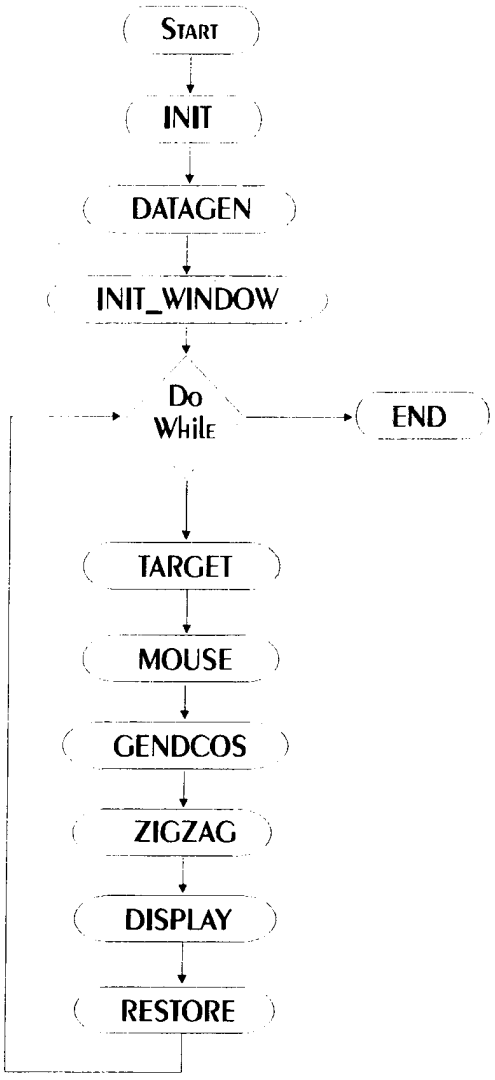
4. Man Pages, Silicon Graphics Inc.

5. Shirely Dowdy and Stanley Weardon, Statistics for Research, John Wiley & Sons, Inc., New York, New York, 1983.

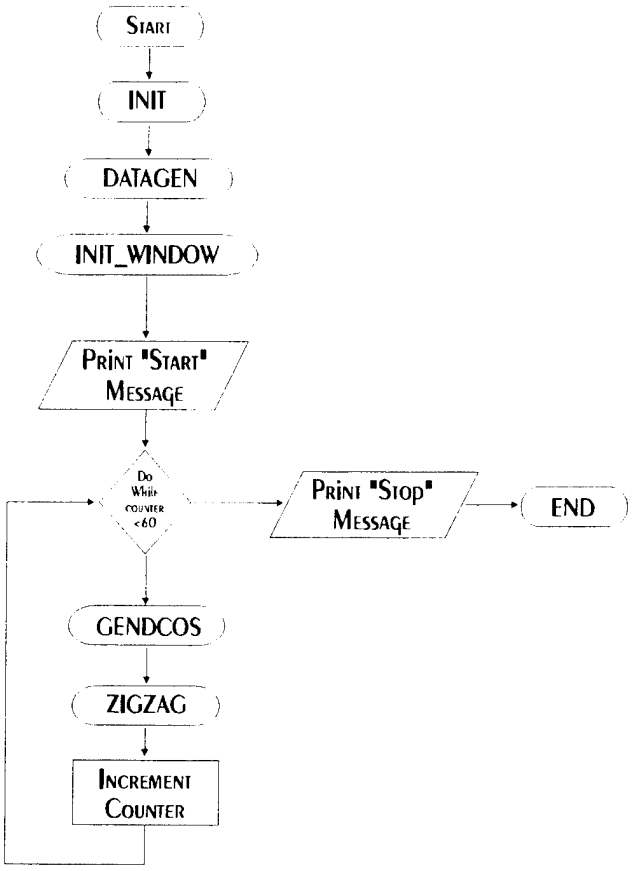
6. Steven J. Zaloga, Modern Soviet Armor, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1979.

APPENDIX A

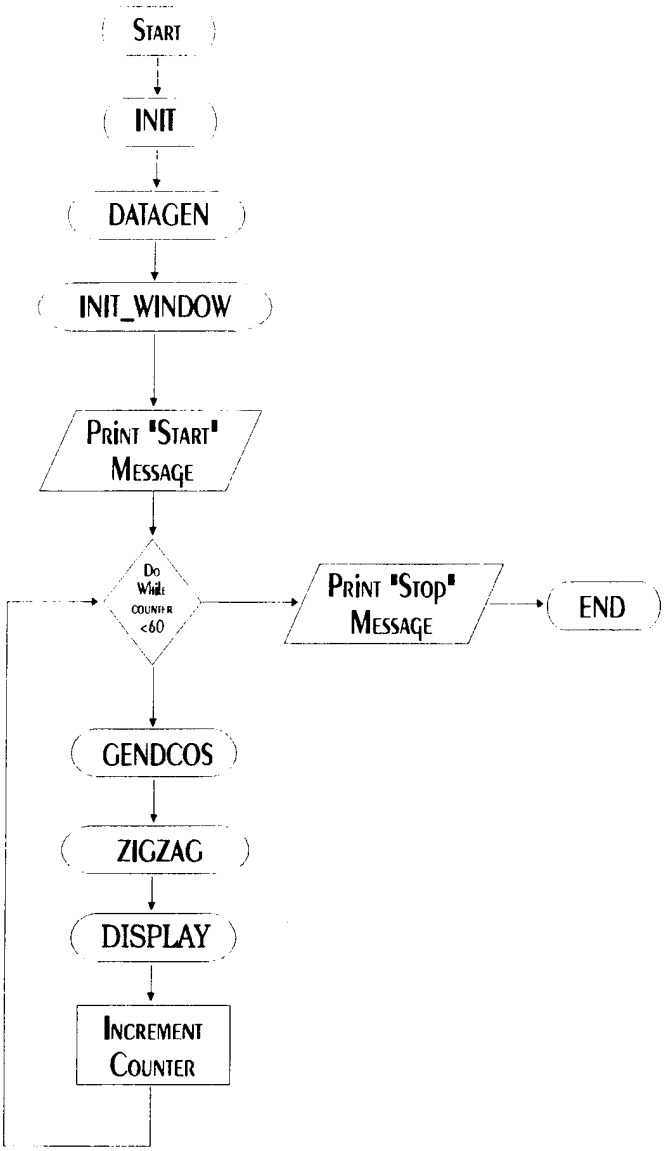
PVG-MAIN



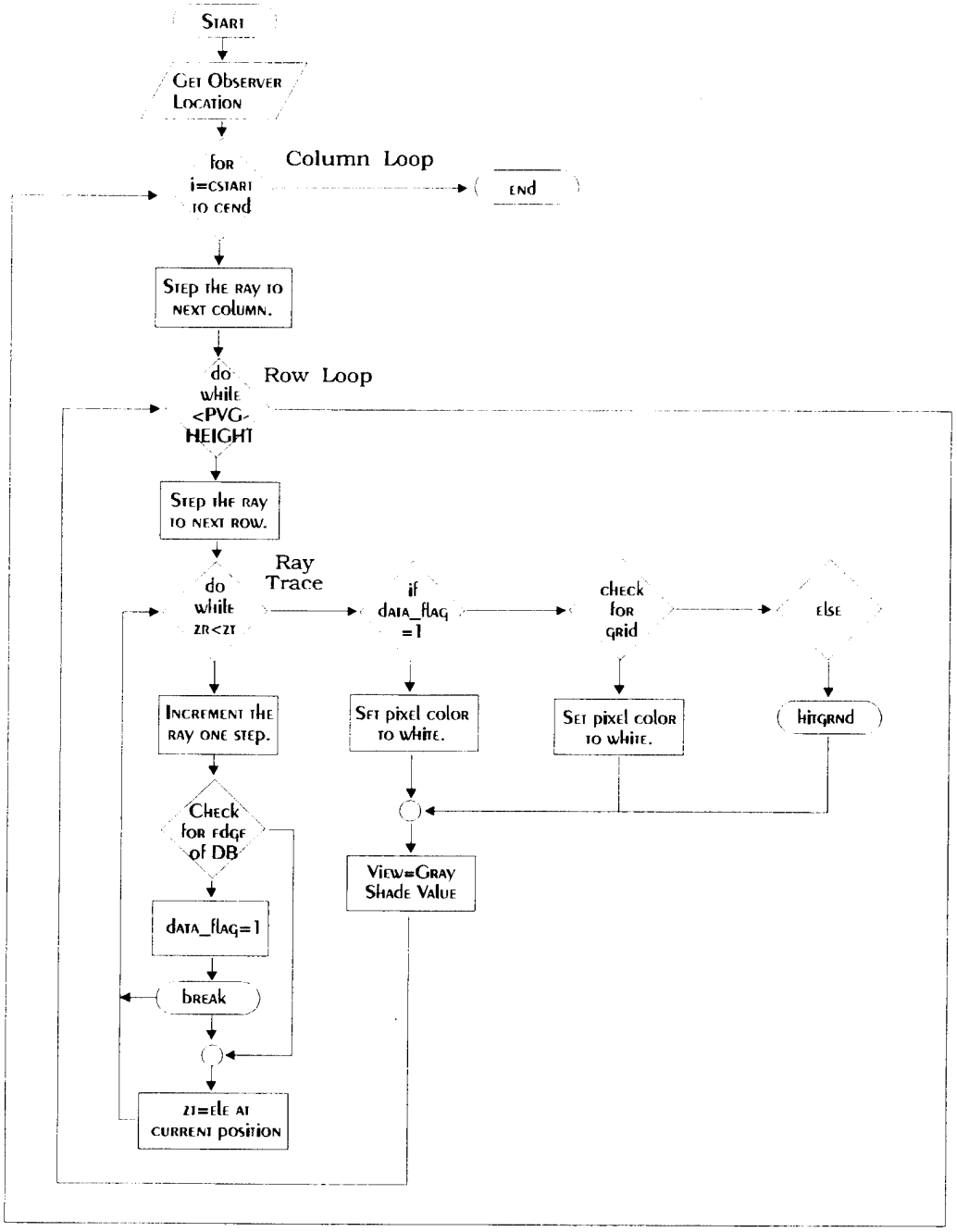
PVG-MAIN (No Video)

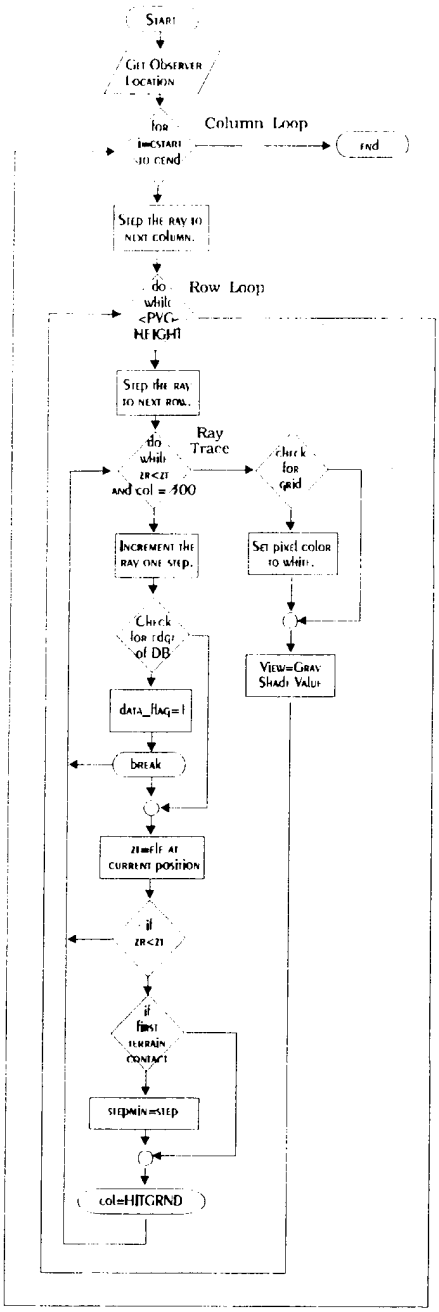


PVG-MAIN (No Mouse)



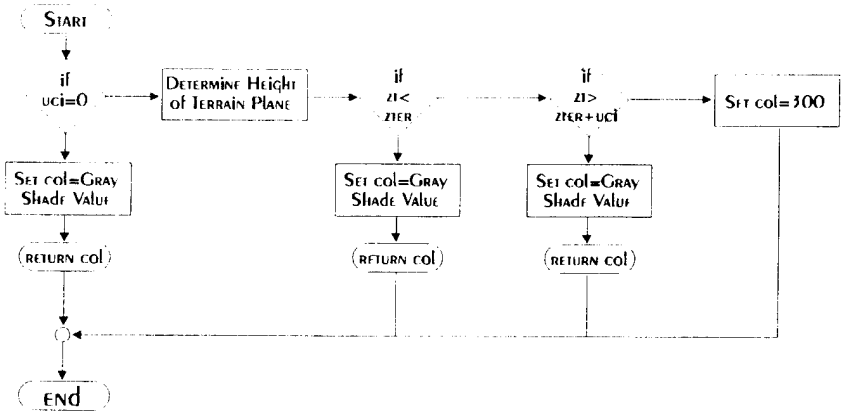
ZIGZAG.C


START
GET OBSERVER LOCATION
for i=CSTART TO CEND
Column Loop
END
STEP THE RAY TO NEXT COLUMN.
do while <PVG-HEIGHT
Row Loop
STEP THE RAY TO NEXT ROW.
do while ZR<ZT
Ray Trace
if data_flag =1
check for grid
ELSE
INCREMENT THE RAY ONE STEP.
SET pixel color TO WHITE.
SET pixel color TO WHITE.
HITGRND
Check for edge of DB
VIEW=GRAY Shade Value
data_flag=1
break
ZT=ELE AT CURRENT POSITION

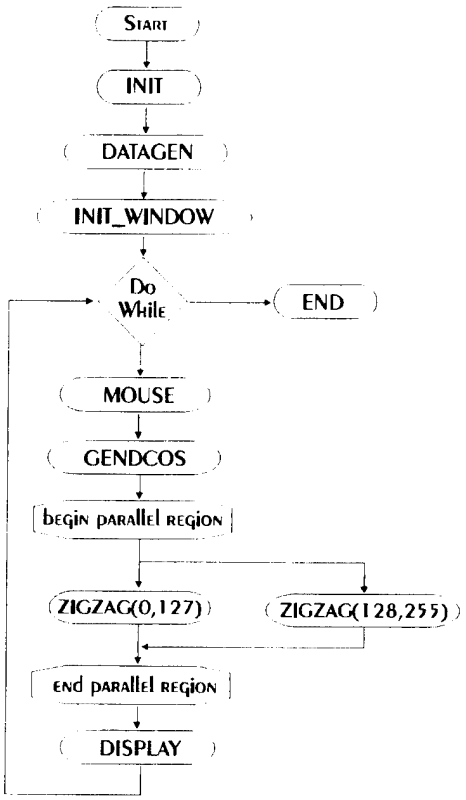

START
GET OBSERVER LOCATION
for i=START to END
Column Loop
END
STEP THE RAY TO NEXT COLUMN.
do while <PYC-HEIGHT
Row Loop
STEP THE RAY TO NEXT ROW.
do while ZR<ZT and col = 400
Ray Trace
check for grid
INCREMENT THE RAY ONE STEP.
Set pixel color to white.
Check for edge of DB
VIEW=GRAY SHADE VALUE
data_flag=1
break
ZR=ELEV AT CURRENT POSITION
if ZR<ZT
if first TERRAIN CONTACT
STEPMIN=STEP
col=HITGRND

ZIGZAGA.C

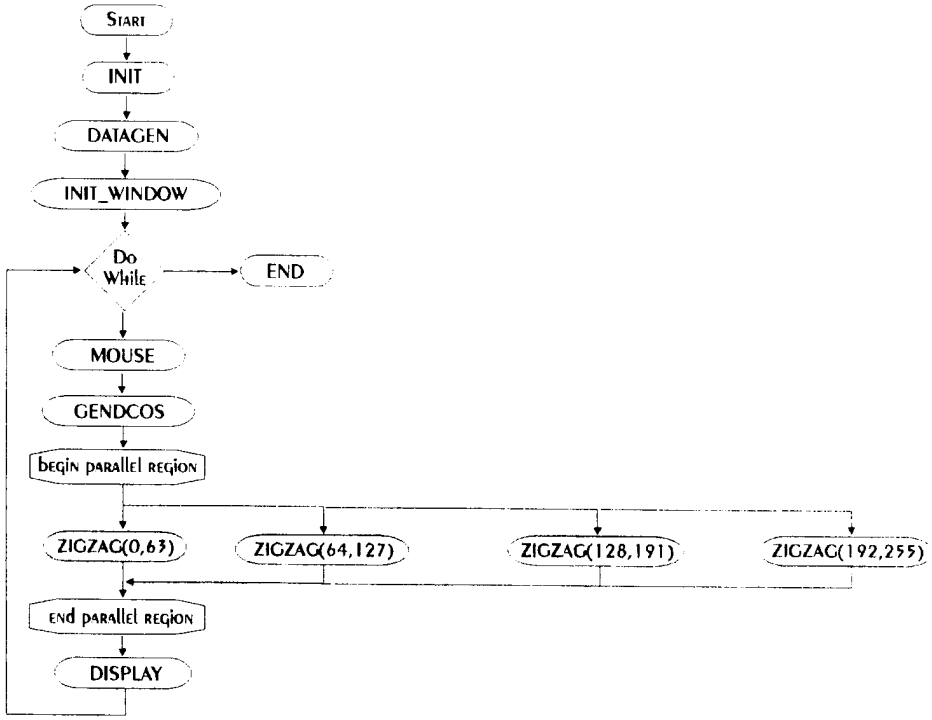
HITGRNDA

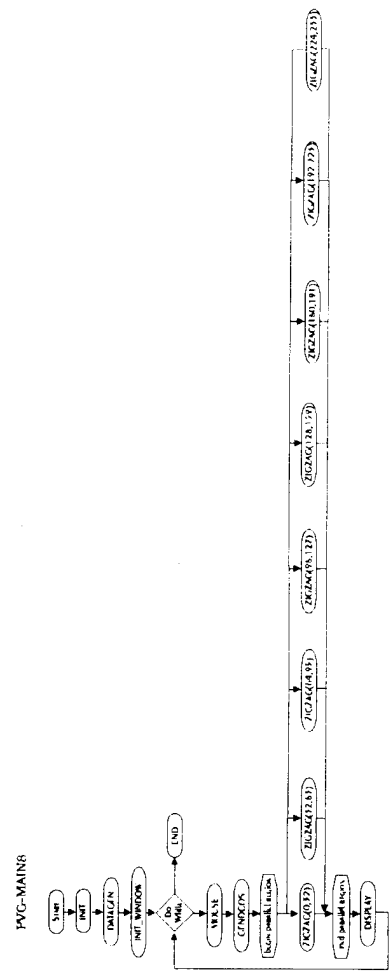


PVG-MAIN2



PVG-MAIN4


START
INIT
DATAGEN
INIT_WINDOW
Do While
END
MOUSE
GENDCOS
begin parallel region
ZIGZAG(0,63)
ZIGZAG(64,127)
ZIGZAG(128,191)
ZIGZAG(192,255)
end parallel region
DISPLAY


PVG-MAIN8
Start
INIT
DATA GEN
INIT_WINDOW
Do Wait
END
VRCSE
GENDCOS
Begin parallel action
ZIGZAG(0,32)
ZIGZAG(32,63)
ZIGZAG(64,95)
ZIGZAG(96,127)
ZIGZAG(128,159)
ZIGZAG(160,191)
ZIGZAG(192,223)
ZIGZAG(224,255)
End parallel action
DISPLAY

APPENDIX B

A. PVG-MAIN.C (FIRST TEST VERSION)

```
/*

        PVG-MAIN.C program

*/

#include<stdio.h>
#include"PVG_DEC.IN"

int n;
        main()
        {
        init();
        datagen();
        init_window();
n=0;
printf("start\n");
        do
        {
        genddcos();
        zigzag(0,255);
        display();
n++;
        }
        while(1);
printf("stop\n");
        }
```

B. PVG-NOMO.C (VERSION 2)

```
/*

        PVG-MAIN.C program

*/

#include<stdio.h>
#include"PVG_DEC.IN"


        main()
        {
int n;
        n=0;
        init();
        datagen();
        printf("start\n");

        do
        {
        genddcos();
        zigzag(0,255);
        display();
        n++;
        }
        while(n<60);
        printf("stop");
        }
```

C. PVG-NOV.C

```
/*

        PVG-MAIN.C program

*/

#include<stdio.h>
#include"PVG_DEC.IN"


        main()
        {
int n;
        n=0;
        init();
        datagen();
        printf("start\n");

        do
        {
        genddcos();
        zigzag(0,255);
        display();
        n++;
        }
        while(n<60);
        printf("stop");
        }
```

D. INIT.C

```
/*

        Program INIT.C to initialize
        global variables

*/

#include<stdio.h>
#include"PVG_DEC.IN"

        int IFOVNOW[10];
        int RAY_SEG[4], MAG;
        int TAR_X,TAR_Y;

        init()
        {

        printf("Magnification (int) ..? ");
        scanf(" %d", &MAG);

/*      View vector         */
        IFOVNOW[0]=0;
        IFOVNOW[1]=0;
        IFOVNOW[2]=700;
        IFOVNOW[3]=30*(int)D2MR;
        IFOVNOW[4]=-40*(int)D2MR;
        IFOVNOW[5]=0*(int)D2MR;
        IFOVNOW[6]=20*(int)D2MR;
        IFOVNOW[7]=0;
        IFOVNOW[8]=0;
        IFOVNOW[9]=0;
        RAY_SEG[0]=128;
        RAY_SEG[1]=128;
        RAY_SEG[2]=0;
        RAY_SEG[3]=0;

/*      Target Start Posistion.  */
        TAR_X=500;
        TAR_Y=500;
}
```

E. CALL_MAP.C

```
/*

        Program CALL_MAP.C to initialize
        the MAP (plan) window

*/

#include<stdio.h>
#include<gl/gl.h>
#include"PVG_DEC.IN"

call_map()
{
        extern int DAT[SCALE][SCALE];
        extern long int MAPWIN_ID, MAINWIN_ID;
        int i, j, col, step;
        unsigned long int viewmap[512*512];

        step=SCALE/512;
        for(i=0;i<512;i++)
        {
                for(j=0;j<512;j++)
          {
                col=(DAT[i*step][j*step]&GSV_MASK);
            if (((int)i*step%256==0) || ((int)j*step%256==0))
                col = 230;
                viewmap[(j*512+i)] = 65793*col*3;
         }
        }

                winset(MAPWIN_ID);
                lrectwrite(0,0,512-1,512-1,viewmap);
                winset(MAINWIN_ID);
}
```

F. GET_BLOCK.C

```
/*
        Program GET_BLOCK.C gets a 256 x 256 block of
        data from the PVDB
*/

#include<stdio.h>
#include<stdlib.h>
#include"PVG_DEC.IN"

int DAT[SCALE][SCALE];

        get_block(int p, int q, int poffset, int qoffset)
        {

        extern int RES;

        FILE *fp;
        int  fret, rret, datv, i, j, ii, jj, xp, yp, block_number, b_res;
        long int offset, block_offset;

/*      1m resolution database generation                       */

        if(RES==1)
         {

/*      printf("mm=%d nn=%d \n", poffset, qoffset);*/
        b_res=256;
        fp=fopen("pvdb.1.33", "rb");
        block_number=16*(p+poffset)+(q+qoffset);
        printf("loading block %d \n", block_number);
        block_offset=4*b_res*b_res*block_number;
                for(i=0;i<b_res;i++)
                { offset=block_offset+4*b_res*i;
                 fret=fseek(fp, offset, 0);
                        for(j=0; j<b_res; j++)
                          {
                        rret=fread(&datv, sizeof(datv), 1, fp);
                        xp=i+(b_res*poffset);
                        yp=j+(b_res*qoffset);
                        DAT[xp][yp]=datv;
```

```
                }
            }
            fclose(fp);
        }

/*      4m resolution database generation      */

        if(RES==4)
          {
        b_res=64;
        fp=fopen("pvdb.4.33", "rb");

        block_number=16*(p+poffset)+(q+qoffset);
        block_offset=4*b_res*b_res*block_number;
            for(i=0;i<b_res;i++)
            { offset=block_offset+4*b_res*i;
             fret=fseek(fp, offset, 0);
                    for(j=0; j<b_res; j++)
                      {
                    fread(&datv, sizeof(datv), 1, fp);
                    xp=4*(i+(b_res*poffset));
                    yp=4*(j+(b_res*qoffset));
                    for(ii=0;ii<4;ii++)
                       {for(jj=0;jj<4;jj++)
                           {DAT[xp+ii][yp+jj]=datv;}
                       }

                      }
            }
            fclose(fp);
        }
        return;
        }
```

G. DATAGEN.C

```
/*      Program DATAGEN.C to generate the data base
            GSV and ELEV from tile33.dat

*/

#include<stdio.h>
#include<stdlib.h>
#include"PVG_DEC.IN"

        int RES=0;

        datagen()
        {

        int x, y, m, n, bl_max;

/*      Get x,y cordinates of lower left data window, and resolution */

        printf("Type in the lower left coordinates of the block (0 to 14) ");
        scanf(" %d %d", &x, &y);

        printf("What resolution map (1 or 4) ");
        scanf( "%d", &RES);

        bl_max=SCALE/256;

/*      Get the LL, UL, LR and UR quadrants of the database      */

        for(m=0;m<bl_max;m++)
          {  for(n=0;n<bl_max;n++)
              {get_block(x, y, m, n);}

          }

        }
```

H. INIT_WINDOW.C

```
/*

        Program INIT_WINDOW.C to initialize
        the GL graphics window

*/

#include<stdio.h>
#include"PVG_DEC.IN"
#include<gl/gl.h>

long    MAINWIN_ID;

        init_window()
        {

        extern int MAG;

        prefsize(PVG_HEIGHT*MAG,PVG_WIDTH*MAG);
        MAINWIN_ID=winopen("PVG-MAIN");
        color(BLACK);
        clear();
        RGBmode();
        gconfig();

        }
```

I. MOUSE.C

```
/*

        Program MOUSE.C to obtain data from the
        mouse and use it to change the
        IFOVNOW vector.

        Mouse controls:

        Bottom left of screen = no viewpoint movement
        cursor up     =     move north
        cursor right  =     move east
        left button   =     reverses north, east displacements
        centre button       =     viewpoint elevation increased
        centre & left       =     viewpoint elevation decreased
        right button  =     positive z rotation
        right & left  =     negative z rotation

*/

#include<stdio.h>
#include"PVG_DEC.IN"
#include<gl/gl.h>
#include<gl/device.h>


        mouse()
        {

        extern int IFOVNOW[10],DAT[SCALE][SCALE];
        extern float trans[3][3];

        int sense;
        float d[3], r[3];

/*      Get displacements                       */
        sense=1;
        d[0]=getvaluator(MOUSEX)/(100*sense);
        d[1]=getvaluator(MOUSEY)/(100*sense);
        if (getbutton(MIDDLEMOUSE)) {d[2]=+10;}
```

```
/*      Get rotations                                   */
        if (getbutton(RIGHTMOUSE))  {r[0]=-10;}

/*      Check sense                                     */
        if (getbutton(LEFTMOUSE))
                {d[0]=-d[0]; d[1]=-d[1]; d[2]=-d[2]; r[0]=-r[0];}

/*      Update the IFOVNOW vector                       */
        IFOVNOW[0]=IFOVNOW[0]+d[0];
        IFOVNOW[1]=IFOVNOW[1]+d[1];
        IFOVNOW[2]=IFOVNOW[2]+d[2];
/*
IFOVNOW[2]=((DAT[IFOVNOW[0]][IFOVNOW[1]]&ELEV_MASK)>>21)+10;
*/
        IFOVNOW[3]=IFOVNOW[3]+r[0];

        }
```

J. GENDDCOS.C

```
/*
        Program GENDDCOS.C generates the direction
        cosines for the ray trace

*/

#include <math.h>
#include "PVG_DEC.IN"

float ddcos[3][3], trans[3][3];

genddcos()
{

float vec[3],borvec[3],rowvec[3],colvec[3];
float h,p,r;
int i,ii;

extern int IFOVNOW[10];
extern int RAY_SEG[4];

/* declare local subroutines used*/
void veh2utm_matrix();
void matnmul();

/* calculate the boresight direction components */
/* set up a unit 1-meter boresight vector in camera coordinates*/
vec[0]=0.;/* column direction */
vec[1]=1.;/* boresight direction*/
vec[2]=0.;/* row direction*/

/* calculate the transform from camera to utm coordinates*/
h=MR2D*IFOVNOW[3];/* heading */
p=MR2D*IFOVNOW[4];/* pitch */
r=MR2D*IFOVNOW[5];/* roll */

veh2utm_matrix(trans,h,p,r);

/* now rotate the vector into utm coordinates */
matnmul(trans,vec,borvec,3,3,1);
```

```
/* now get the unit pixel vector in the column direction */
/* set up a unit pixel column vector in camera coordinates*/
vec[0]=((float)IFOVNOW[6])/(1000.0*(float)PVG_WIDTH);
                    /* column direction */
vec[1]=0.;/* borsight direction*/
vec[2]=0.;/* row direction*/

/* now rotate the vector into utm coordinates */
matnmul(trans,vec,colvec,3,3,1);

/* now get the unit pixel vector in the row direction */
/* set up a unit row vector in camera coordinates*/
vec[0]=0.;/* column direction */
vec[1]=0.;/* boresight direction*/
vec[2]=((float)IFOVNOW[6])/(1000.0*(float)PVG_HEIGHT);
            /*row direction */

/* now rotate the vector into utm coordinates */
matnmul(trans,vec,rowvec,3,3,1);
/* scale the row column vectors by the image scale*/
for (i=0;i<3;i++)
      {
      ddcos[1][i]=(rowvec[i]);
      ddcos[2][i]=(colvec[i]);
      ddcos[0][i]=borvec[i]
            +ddcos[1][i]*(RAY_SEG[0]-(PVG_HEIGHT/2))
            +ddcos[2][i]*(RAY_SEG[1]-(PVG_WIDTH/2));
     }
}
```

K. MATNMUL.C

```
/*
        Program MATNMUL.C to multiply
        two matrices

USE EXAMPLE:
matnmul(mat1, mat2, mat3, dim1, dim2, dim3);

dimension of mat1 is dim1 x dim2
dimension of mat2 is dim2 x dim3
dimension of mat3 is dim1 x dim3
*/

 matnmul(mat1,mat2,mat3,dim1,dim2,dim3)

float *mat1, *mat2, *mat3;
int dim1, dim2, dim3;

{
  int i, j, k;
  float x;

  for (j=0; j<dim1; j++)
  for (k=0; k<dim3; k++)
    *(mat3+j*dim3+k) = 0.0;
  for (k=0; k<dim3; k++)
  for (j=0; j<dim1; j++)
  for (i=0; i<dim2; i++)
    *(mat3+j*dim3+k) += *(mat1+j*dim2+i)*(*(mat2+i*dim3+k));
}
```

L. VEH2UTM_MATRIX.C

```
/*
      PROGRAM VEH2UTM_MATRIX.C generates a
      3x3 transform matrix to convert vectors in
      vehicle coordinates to UTM components
*/

#include <math.h>
#define PI 3.14159265

 veh2utm_matrix(trans, h, p, r)

float *trans;/* output transform matrix */
float h, p, r;/* heading, pitch roll in degrees are input*/

{
 float hm[3][3], pm[3][3], rm[3][3], am[3][3];
 float cos_h, sin_h;
 float cos_p, sin_p;
 float cos_r, sin_r;
 int j;

 cos_h = cos((float) h*PI/180.0);
 sin_h = sin((float) h*PI/180.0);

 cos_p = cos((float) p*PI/180.0);
 sin_p = sin((float) p*PI/180.0);

 cos_r = cos((float) r*PI/180.0);
 sin_r = sin((float) r*PI/180.0);

 hm[0][0] =  cos_h;   hm[0][1] =  sin_h;    hm[0][2] =   0.0;
 hm[1][0] = -sin_h;   hm[1][1] =  cos_h;    hm[1][2] =   0.0;
 hm[2][0] =   0.0;    hm[2][1] =   0.0;     hm[2][2] =   1.0;

 pm[0][0] =   1.0;    pm[0][1] =   0.0;     pm[0][2] =   0.0;
 pm[1][0] =   0.0;    pm[1][1] =  cos_p;    pm[1][2] = -sin_p;
 pm[2][0] =   0.0;    pm[2][1] =  sin_p;    pm[2][2] =  cos_p;

 rm[0][0] =  cos_r;   rm[0][1] =   0.0;     rm[0][2] =  sin_r;
 rm[1][0] =   0.0;    rm[1][1] =   1.0;     rm[1][2] =   0.0;
```

```
  rm[2][0] = -sin_r;    rm[2][1] =    0.0;      rm[2][2] =  cos_r;

  matnmul(pm,rm,am,3,3,3);
  matnmul(hm,am,trans,3,3,3);

}
```

M. ZIGZAG.C

```
/*      PROGRAM ZIGZAG.C

        Basic ray trace algorithm.  This version includes the step back 3, then
restart.

*/
#include<stdio.h>
#include<stdlib.h>
#include"PVG_DEC.IN"

extern int hitgrnd(int, int, int, int, float, float, float, int);
extern float ddcos[3][3];
extern long int IFOVNOW[10], DAT[SCALE][SCALE];

int VIEW[PVG_WIDTH*PVG_HEIGHT];

void zigzag(int, int);

void zigzag(int cstart, int cend)
{
   register int    step, stepmin;
   register float  e, n, zr, de, dn, dz;
   long        zt=0;
   int         col;
   int         i, j;
   int         data_flag=0;
   float       idcos20, idcos21, idcos22;
   const float deb = ddcos[0][0];          /* boresight direction cosines */
   const float dnb = ddcos[0][1];
   const float dzb = ddcos[0][2];
   const float dcos10 = ddcos[1][0];
   const float dcos11 = ddcos[1][1];
   const float dcos12 = ddcos[1][2];
   const float dcos20 = ddcos[2][0];
   const float dcos21 = ddcos[2][1];
   const float dcos22 = ddcos[2][2];

   step = stepmin = ray_num = 0;        /* initialize these variables */

   e  = (float)IFOVNOW[0];              /* define start location */
```

```
    n  = (float)IFOVNOW[1];
    zr = (float)IFOVNOW[2];

    for(i=cstart; i<cend; i++)
     {
        idcos20 = (float)i*dcos20;      /* calc. these as few times as nec. */
        idcos21 = (float)i*dcos21;
        idcos22 = (float)i*dcos22;

        j = 0;                          /* start row loop */
        do {

           de = deb + (float)j*dcos10 + idcos20;
           dn = dnb + (float)j*dcos11 + idcos21;
           dz = dzb + (float)j*dcos12 + idcos22;

           do {
              step++;
              zr = (float)IFOVNOW[2] + (float)step*dz;
              e  = (float)IFOVNOW[0] + (float)step*de;
              n  = (float)IFOVNOW[1] + (float)step*dn;

              if ((e>(float)SCALE) || (n>(float)SCALE)) {
                 data_flag = 1;
                 break;
              }

              zt = (int)(DAT[(int)e][(int)n]&ELEV_MASK)>>21;

           } while (zr > (float)zt);

           if ( data_flag ) {                    /* out of bounds test */
              col = 255; }

           /* draw 256x256 white grid */
           else if (((int)e%256==0) || ((int)n%256==0))
              col = 255;
           else
{          col = hitgrnd((int)e, (int)n, (int)zr, zt, de, dn, dz, step);
           VIEW[(j*PVG_WIDTH+i)] = 65793*col*3; }

           data_flag =  0;              /* clear flags */
```

```
            if(j == 0)
                stepmin = step;

            step -= 3;
        } while (++j < PVG_HEIGHT);        /* end of column loop */

        step = stepmin - 3;
    }              /* end of row loop */

    if (ray_num>0) {fprintf(stderr, "ray hits = %d \n", ray_num);}

}
```

N. ZIGZAGA.C

```
/*      PROGRAM ZIGZAG.C

        Basic ray trace algorithm.  This version includes the step back 3, then
restart.

*/
#include<stdio.h>
#include<stdlib.h>
#include"PVG_DEC.IN"

extern int hitgrnd(int, int, int, int);
extern float ddcos[3][3];
extern long int IFOVNOW[10], DAT[SCALE][SCALE];

int VIEW[PVG_WIDTH*PVG_HEIGHT];

void zigzag(int, int);

void zigzag(int cstart, int cend)
{
   register int    step, stepmin;
   register float  e, n, zr, de, dn, dz;
   long        zt=0;
   int         col;
   int         i, j;
   int         data_flag=0;
   float       idcos20, idcos21, idcos22;
   const float deb = ddcos[0][0];          /* boresight direction cosines */
   const float dnb = ddcos[0][1];
   const float dzb = ddcos[0][2];
   const float dcos10 = ddcos[1][0];
   const float dcos11 = ddcos[1][1];
   const float dcos12 = ddcos[1][2];
   const float dcos20 = ddcos[2][0];
   const float dcos21 = ddcos[2][1];
   const float dcos22 = ddcos[2][2];

   step = stepmin = 0;       /* initialize these variables */

   e  = (float)IFOVNOW[0];                 /* define start location */
```

```
n  = (float)IFOVNOW[1];
zr = (float)IFOVNOW[2];

for(i=cstart; i<cend; i++)
 {
    idcos20 = (float)i*dcos20;      /* calc. these as few times as nec. */
    idcos21 = (float)i*dcos21;
    idcos22 = (float)i*dcos22;

    j = 0;                          /* start row loop */
    do {

       de = deb + (float)j*dcos10 + idcos20;
       dn = dnb + (float)j*dcos11 + idcos21;
       dz = dzb + (float)j*dcos12 + idcos22;

       do {
          step++;
          zr = (float)IFOVNOW[2] + (float)step*dz;
          e  = (float)IFOVNOW[0] + (float)step*de;
          n  = (float)IFOVNOW[1] + (float)step*dn;

          if ((e>(float)SCALE) || (n>(float)SCALE)) {
             col=255;
             break;
          }

       zt = (int)(DAT[(int)e][(int)n]&ELEV_MASK)>>21;

/*     Check for first terrain contact, and call hitgrnd  */
       if (zr < (float)zt)
          { if (stepmin==0) stepmin=step;
            col = hitgrnd((int)e, (int)n, (int)zr, (int)zt);
           }

       } while ((zr > (float)zt) || (col == 300));

       if (((int)e%256==0) || ((int)n%256==0))
          col = 255;
       VIEW[(j*PVG_WIDTH+i)] = 65793*col*3;

       step = stepmin - 3;
```

```
            stepmin = 0;

        } while (++j < PVG_HEIGHT);          /* end of column loop */
        step = 0;
    }               /* end of row loop */

}
```

O. HITGRNDA.C

```
/*

        Program HITGRND.C to determine the
        gray scale when ray is beneath ELEV.

*/

#include<stdio.h>
#include"PVG_DEC.IN"

        int hitgrnd(int ei, int ni, int zri, int zt)
        {

extern  int IFOVNOW[10], DAT[SCALE][SCALE];


        int shadow, zter, col;

/*      Check for bald terrain                  */
        if ((DAT[ei][ni]&UCI_MASK)==0)
                {col=(DAT[ei][ni]&GSV_MASK); return(col);}

/*      Overhang, check if terrain is hit       */
        zter=zt-((DAT[ei][ni]&VGH_MASK)>>10);
        if(zri<zter)
          {shadow = (DAT[ei][ni]&GSV_MASK) - 50;
           if ( shadow < 0 ) shadow = 0;
           col = shadow; return(col);}

/*      If not, check if we are in the overhang */
        if(zri>(zter+(DAT[ei][ni]&UCI_MASK)>>18))
                {col=(DAT[ei][ni]&GSV_MASK); return(col);}

/*      if not, we are under the overhang, continue ray trace   */
        col = 300;return(col);

        }
```

P. GRND_INSERT.C

```
/*  determine the gray scale when ray is beneath ELEV. */

        int shadow, zter;

        shadow = 0;
/*      Check for bald terrain                          */
        if ((DAT[e][n]&UCI_MASK) = = 0)
                {col = (DAT[e][n]&GSV_MASK);}

/*      Overhang, check if terrain is hit               */
        zter = zt-((DAT[e][n]&VGH_MASK) > > 10);
        if(zr < zter)   {col = shadow;}

/*      If not, check if we are in the overhang */
        if(zr > (zter + (DAT[e][n]&UCI_MASK) > > 18))
                {col = (DAT[e][n]&GSV_MASK);}
```

Q. DISPLAY.C

```
/*      PROGRAM DISPLAY.C

        Puts array VIEW on the screen

        input parameters: VIEW [PVG_HEIGHT*PVG_WIDTH]
        output parameters: none
        global parameters used: SCALE

 */

#include<stdio.h>
#include<gl/gl.h>
#include"PVG_DEC.IN"


        display()
        {

extern          unsigned long int VIEW[PVG_HEIGHT*PVG_WIDTH];
extern  int MAG;

int i,j;

/*extern long int VIEW[PVG_HEIGHT*PVG_WIDTH];*/

/*------------  Display section -----------------------*/

        rectzoom((float)1*MAG,(float)1*MAG);
        lrectwrite(0,0,PVG_HEIGHT-1,PVG_WIDTH-1,VIEW);

        }
```

R. TARGETA.C

```
/*
        Program TARGET.C modifies the PVG data
        base to place the target(s)

*/

#include<stdio.h>
#include<stdlib.h>
#include"PVG_DEC.IN"
#include<gl/gl.h>

unsigned long SAVECOL[24];

        target()
        {

        extern long int MAINWIN_ID;
        extern int DAT[SCALE][SCALE];
        extern int TAR_X,TAR_Y;
        int i, j, xmin, ymin, xmax, ymax, data, step;

        short int redvector[3]={255, 0, 0};
        short int bluevector[3]={0, 0, 255};

/*      move target                     */
        TAR_X=TAR_X+1;

/*      step=SCALE/512; */

/*      Save ground terrain color array */
        for(i=TAR_X;i<TAR_X+7;i++)
                {for(j=TAR_Y;j<TAR_Y+3;j++)
                  {SAVECOL[3*(i-TAR_X)+(j-TAR_Y)]=DAT[i][j]&GSV_MASK;}
                }
/*      change elevations, change color and clear nature bit            */

        j=TAR_Y;
        for(i=TAR_X+1;i<TAR_X+3;i++)
                {DAT[i][j]=DAT[i][j]+0x00100000;
                 DAT[i][j]=DAT[i][j]&(~NATURE_MASK);
```

```
        DAT[i][j] = DAT[i][j]&(GSV_CLEAR);
        DAT[i][j] = DAT[i][j] + 0x0000000a;}
DAT[TAR_X + 3][j] = DAT[i][j] + 0x00200000;
DAT[TAR_X + 3][j] = DAT[i][j]&(~NATURE_MASK);
DAT[TAR_X + 3][j] = DAT[i][j]&(GSV_CLEAR);
DAT[TAR_X + 3][j] = DAT[i][j] + 0x00000005;
for(i = TAR_X + 4;i < TAR_X + 7;i + +)
        {DAT[i][j] = DAT[i][j] + 0x00100000;
         DAT[i][j] = DAT[i][j]&(~NATURE_MASK);
         DAT[i][j] = DAT[i][j]&(GSV_CLEAR);
         DAT[i][j] = DAT[i][j] + 0x0000000a;}
j = TAR_Y + 1;
DAT[TAR_X][j] = DAT[i][j] + 0x00200000;
DAT[TAR_X][j] = DAT[i][j]&(~NATURE_MASK);
DAT[TAR_X][j] = DAT[i][j]&(GSV_CLEAR);
DAT[TAR_X][j] = DAT[i][j] + 0x00000005;

DAT[TAR_X + 1][j] = DAT[i][j] + 0x00100000;
DAT[TAR_X + 1][j] = DAT[i][j]&(~NATURE_MASK);
DAT[TAR_X + 1][j] = DAT[i][j]&(GSV_CLEAR);
DAT[TAR_X + 1][j] = DAT[i][j] + 0x0000000a;

for(i = TAR_X + 3;i < TAR_X + 6;i + +)
        {DAT[i][j] = DAT[i][j] + 0x00200000;
         DAT[i][j] = DAT[i][j]&(~NATURE_MASK);
         DAT[i][j] = DAT[i][j]&(GSV_CLEAR);
         DAT[i][j] = DAT[i][j] + 0x00000005;}

DAT[TAR_X + 6][j] = DAT[i][j] + 0x00100000;
DAT[TAR_X + 6][j] = DAT[i][j]&(~NATURE_MASK);
DAT[TAR_X + 6][j] = DAT[i][j]&(GSV_CLEAR);
DAT[TAR_X + 6][j] = DAT[i][j] + 0x0000000a;

j = TAR_Y + 2;
for(i = TAR_X + 1;i < TAR_X + 3;i + +)
        {DAT[i][j] = DAT[i][j] + 0x00100000;
         DAT[i][j] = DAT[i][j]&(~NATURE_MASK);
         DAT[i][j] = DAT[i][j]&(GSV_CLEAR);
         DAT[i][j] = DAT[i][j] + 0x0000000a;}

DAT[TAR_X + 3][j] = DAT[i][j] + 0x00200000;
DAT[TAR_X + 3][j] = DAT[i][j]&(~NATURE_MASK);
```

```
DAT[TAR_X + 3][j] = DAT[i][j]&(GSV_CLEAR);
DAT[TAR_X + 3][j] = DAT[i][j] + 0x00000005;

for(i = TAR_X + 4;i < TAR_X + 7;i + +)
        {DAT[i][j] = DAT[i][j] + 0x00100000;
         DAT[i][j] = DAT[i][j]&(~NATURE_MASK);
         DAT[i][j] = DAT[i][j]&(GSV_CLEAR);
         DAT[i][j] = DAT[i][j] + 0x0000000a;}

}
```

S. RESTOREA.C

```
/*
        Program RESTORE.C to restore the PVG data
        base to remove the target(s)

*/

#include<stdio.h>
#include<stdlib.h>
#include"PVG_DEC.IN"

        restore()
        {
        extern DAT[SCALE][SCALE];
        extern TAR_X, TAR_Y;
        extern SAVECOL[24];

/*      int step=SCALE/512; */
        int i,j;

/*      change elevations and set nature bit          */

        j=TAR_Y;
        for(i=TAR_X+1;i<TAR_X+3;i++)
                {DAT[i][j]=DAT[i][j]-0x00100000;
                 DAT[i][j]=DAT[i][j] | NATURE_MASK;}

        DAT[TAR_X+3][j]=DAT[i][j]-0x00200000;
        DAT[TAR_X+3][j]=DAT[i][j] | NATURE_MASK;

        for(i=TAR_X+4;i<TAR_X+7;i++)
                {DAT[i][j]=DAT[i][j]-0x00100000;
                 DAT[i][j]=DAT[i][j] | NATURE_MASK;}

        j=TAR_Y+1;
        DAT[TAR_X][j]=DAT[i][j]-0x00200000;
        DAT[TAR_X][j]=DAT[i][j] | NATURE_MASK;
        DAT[TAR_X+1][j]=DAT[i][j]-0x00100000;
        DAT[TAR_X+1][j]=DAT[i][j] | NATURE_MASK;
        for(i=TAR_X+3;i<TAR_X+6;i++)
                {DAT[i][j]=DAT[i][j]-0x00200000;
```

```
            DAT[i][j] = DAT[i][j] | NATURE_MASK;}
        DAT[TAR_X + 6][j] = DAT[i][j]-0x00100000;
        DAT[TAR_X + 6][j] = DAT[i][j] | NATURE_MASK;
        j = TAR_Y + 2;
        for(i = TAR_X + 1;i < TAR_X + 3;i + +)
            {DAT[i][j] = DAT[i][j]-0x00100000;
             DAT[i][j] = DAT[i][j] | NATURE_MASK;}
        DAT[TAR_X + 3][j] = DAT[i][j]-0x00200000;
        DAT[TAR_X + 3][j] = DAT[i][j] | NATURE_MASK;
        for(i = TAR_X + 4;i < TAR_X + 7;i + +)
            {DAT[i][j] = DAT[i][j]-0x00100000;
             DAT[i][j] = DAT[i][j] | NATURE_MASK;}

/*      reset color  */

        for(i = TAR_X;i < TAR_X + 7;i + +)
            {for(j = TAR_Y;j < TAR_Y + 3;j + +)
              {DAT[i][j] = DAT[i][j]&GSV_CLEAR;
               DAT[i][j] = DAT[i][j] + SAVECOL[3*(i-TAR_X) + (j-TAR_Y)];}
            }
printf("\n");

        }
```

T. TESTFN5G.C

```
/*  This is a test function designed to assist in exploring the idiosincracies
    of the IRIS Power C Analyzer.                                          */

int     i,j,k,m,n,p,q,r;        /* Indices used in 'for' loops. */
float   alt1,alt2,alt3,alt4,    /* height obove the block. */
        grnd;                   /* tollerance for height of zero. */

float   cell1[1000][1000],          /* all the cells of a block */
        cell2[1000][1000],
        cell3[1000][1000],
        cell4[1000][1000];

float blocker ( float alt, float cell[1000][1000] )
{
                for(i=0;i<1000;i++)
                {       for (j=0;j<1000;j++)
                                cell[i][j]=7/alt1;
                }
}


void main (void)
{

/* Initialize varaibles */

        grnd=.0001;
        alt1=5;
        alt2=10;
        alt3=15;
        alt4=20;

/* Begin paralellizable while loops */
#pragma parallel
{
        #pragma independent
        while ( alt1>grnd)
        {       alt1=alt1/2;
                blocker(alt1,cell1);
                if (alt1<grnd)
```

```
                printf("Block1\n");
        }
        #pragma independent
        while ( alt2>grnd)
        {       alt2=alt2/2;
                blocker(alt2,cell2);
                if (alt2<grnd)
                        printf("Block2\n");
        }
        #pragma independent
        while ( alt3>grnd)
        {       alt3=alt3/2;
                blocker(alt3,cell3);
                if (alt3<grnd)
                        printf("Block3\n");
        }
        #pragma independent
        while ( alt4>grnd)
        {       alt4=alt4/2;
                blocker(alt4,cell4);
                if (alt4<grnd)
                        printf("Block4\n");
}
        }
}
```

U. PVG-MAIN2.C

```
/*

        PVG-MAIN.C program

*/

#include<stdio.h>
#include"PVG_DEC.IN"


        main()
        {
int n;
        n=0;
        init();
        datagen();
        printf("start\n");

        do
        {
        mouse();
        genddcos();
#pragma parallel
{
        #pragma independent
        {
                zigzag(0,127);
        }
        #pragma independent
        {
                zigzag(128,255);
        }
}
        display();
        n++;
        }
        while(n<60);
        printf("stop");
        }
```

V. PVG-MAIN4.C

```
/*

        PVG-MAIN.C program

*/

#include<stdio.h>
#include"PVG_DEC.IN"


        main()
        {
int n;
        n=0;
        init();
        datagen();
        printf("start\n");

        do
        {
        mouse();
        genddcos();
#pragma parallel
{
        #pragma independent
        {
                zigzag(0,21);
        }
        #pragma independent
        {
                zigzag(32,63);
        }
        #pragma independent
        {
                zigzag(64,95);
        }
        #pragma independent
        {
                zigzag(96,127);
        }
```

```
        #pragma independent
        {
                zigzag(128,159);
        }
        #pragma independent
        {
                zigzag(160,191);
        }
        #pragma independent
        {
                zigzag(192,223);
        }
        #pragma independent
        {
                zigzag(224,255);
        }

}
        display()
        n++;
        }
        while(n<60);
        printf("stop");
        }
```

W. VTEST.C

```
#include <gl/gl.h>

int n
int blank(10*200)

main()
{     for(n=0;n<2000;n++)
                blank[n]=400;
      prefsize(400,200);
      winopen("green");
      color(GREEN);
      clear();
#pragma parallel
{
      #pragma independent
      { lrectwrite(0,0,9,199,blank); }
      #pragma independent
      { lrectwrite(20,0,29,199,blank); }
      #pragma independent
      { lrectwrite(40,0,49,199,blank); }
      #pragma independent
      { lrectwrite(60,0,69,199,blank); }
}
      sleep(10);
      gexit();
      return 0;
```

APPENDIX C

A. LOG810

1. Testing will begin with phase 1 of test procedure Nr 2 being applied to zigzag.c. Magnification will be set to 2 each time and the starting point will be 0,0. Vehicle 1 will be manuevered as required to keep databse boundaries out of the field of view. This set of trials will alct as the baseline.

```
indy1 23% ls
PVG_DEC.IN      get_block.o     matnmul.u       target.o
call_map.c      get_block.u     mouse.c         target.u
call_map.o      hitgrnd.c       mouse.o         veh2utm_matrix.c
call_map.u      hitgrnd.o       mouse.u         veh2utm_matrix.o
datagen.c       hitgrnd.u       pvdb.1.33       veh2utm_matrix.u
datagen.o       init.c          pvg-main*       zigzag.c
datagen.u       init.o          pvg-main.c      zigzag.o
display.c       init.u          pvg-main.o      zigzag.u
display.o       init_window.c   pvg-main.u      zigzag1.c
display.u       init_window.o   restore.c       zigzag2.c
genddcos.c      init_window.u   restore.o       zigzag3.c
genddcos.o      makefile        restore.u       zigzag4.c
genddcos.u      matnmul.c       spaceball.c
get_block.c     matnmul.o       target.c

indy1 25% prof pvg-main
------------------------------------------------------------------------
Profile listing generated Thu Aug 10 08:45:57 1995
   with:      prof pvg-main
------------------------------------------------------------------------

samples  time   CPU    FPU   Clock   N-cpu  S-interval Countsize
   558   5.6s  R4400  R4010 100.0MHz  1     10.0ms     2(bytes)

Each sample covers 4 bytes for every 10.0ms ( 0.18% of 5.5800sec)


------------------------------------------------------------------------
 -p[rocedures] using pc-sampling.
 Sorted in descending order by the number of samples in each procedure.
 Unexecuted procedures are excluded.
------------------------------------------------------------------------
```

```
samples  time(%)      cum time(%)     procedure (file)

    265  2.6s( 47.5)  2.6s( 47.5)      _read (libc.so.1:read.s)
    193  1.9s( 34.6)  4.6s( 82.1)      fread (libc.so.1:fread.c)
     51  0.51s(  9.1)  5.1s( 91.2)     memcpy (libc.so.1:bcopy.s)
     36  0.36s(  6.5)  5.5s( 97.7)     get_block (pvg-main:get_block.c)
      5  0.05s(  0.9)  5.5s( 98.6)     _open (libc.so.1:open.s)
      3  0.03s(  0.5)  5.5s( 99.1)     __filbuf (libc.so.1:_filbuf.c)
      2  0.02s(  0.4)  5.5s( 99.5)     _lseek (libc.so.1:lseek.s)
      1  0.01s(  0.2)  5.6s( 99.6)     _select (libc.so.1:select.s)
      1  0.01s(  0.2)  5.6s( 99.8)     _close (libc.so.1:close.s)
      1  0.01s(  0.2)  5.6s(100.0) XkbKeysymToModifiers (libX11.so.1:XKBBind.c)

    558  5.6s(100.0)  5.6s(100.0)      TOTAL
```

2. Trial one was conducted over a 40 frame interval. As are all subsequent trials.

Trial Nr 1 , execution time: 5.6s.
Trial Nr 2 , execution time: 19s. ?
Trial Nr 3 , execution time: 19s. ?
Trial Nr 4 , execution time: 5.6s.
Trial Nr 5 , execution time: 5.5s.
Trial Nr 6 , execution time: 5.5s.
Trial Nr 7 , execution time: 5.6s.
Trial Nr 8 , execution time: 5.5s.
Trial Nr 9 , execution time: 5.8s.
Trial Nr 10 , execution time: 5.5s.
Trial Nr 11 , execution time: 5.9s.
Trial Nr 12 , execution time: 5.5s.
Trial Nr 13 , execution time: 5.5s.
Trial Nr 14 , execution time: 5.6s.
Trial Nr 15 , execution time: 5.5s.
Trial Nr 16 , execution time: 5.8s.
Trial Nr 17 , execution time: 5.9s.
Trial Nr 18 , execution time: 5.6s.
Trial Nr 19 , execution time: 5.6s.
Trial Nr 20 , execution time: 5.7s.
Trial Nr 21 , execution time: 5.7s.
Trial Nr 22 , execution time: 5.9s.
Trial Nr 23 , execution time: 6.0s.
Trial Nr 24 , execution time: 5.9s.

Trial Nr 25 , execution time: 5.6s.
Trial Nr 26 , execution time: 5.6s.
Trial Nr 27 , execution time: 5.7s.
Trial Nr 28 , execution time: 5.6s.
Trial Nr 29 , execution time: 5.6s.
Trial Nr 30 , execution time: 19s. ?
Trial Nr 31 , execution time: 6.4s.
Trial Nr 32 , execution time: 5.3s.
Trial Nr 33 , execution time: 19s. ?

3. Phase 2 will be run using a 20 frame interval.

Trial Nr 1, execution time: 36.0.
Trial Nr 2, execution time: 36.2.
Trial Nr 3, execution time: 39.9.
Trial Nr 4, execution time: 35.1.
Trial Nr 5, execution time: 35.1.
Trial Nr 6, execution time: 43.0.
Trial Nr 7, execution time: 38.2.
Trial Nr 8, execution time: 40.0.
Trial Nr 9, execution time: 39.0.
Trial Nr 10, execution time: 40.8.
Trial Nr 12, execution time: 42.3.
Trial Nr 12, execution time: 47.7.
Trial Nr 13, execution time: 50.2.
Trial Nr 14, execution time: 42.6.
Trial Nr 15, execution time: 40.7.
Trial Nr 16, execution time: 35.7.
Trial Nr 17, execution time: 35.3.
Trial Nr 18, execution time: 35.7.
Trial Nr 19, execution time: 35.4.
Trial Nr 20, execution time: 36.2.
Trial Nr 21, execution time: 37.2.
Trial Nr 22, execution time: 36.5.
Trial Nr 23, execution time: 33.5.
Trial Nr 24, execution time: 37.9.
Trial Nr 25, execution time: 42.8.
Trial Nr 26, execution time: 38.5.
Trial Nr 27, execution time: 35.9.
Trial Nr 28, execution time: 35.7.
Trial Nr 29, execution time: 36.1.

B. LOG98A

1. The following are a series of runs using pixie on varius versions of pvg.
The first run is the PVG program with video and mouse.

```
> prof -pixie pvg-V8 pvg-V8.Counts.*
------------------------------------------------------------------------
Profile listing generated Fri Sep  8 13:14:05 1995
   with:      prof -pixie pvg-V8 pvg-V8.Counts.15847 pvg-V8.Counts.15851 pvg-V8.Counts.15853
pvg-V8.Counts.15854    pvg-V8.Counts.15855    pvg-V8.Counts.15856    pvg-V8.Counts.15857
pvg-V8.Counts.15858 pvg-V8.Counts.15859
------------------------------------------------------------------------

Total cycles   Total Time  Instructions Cycles/inst   Clock   Target
 3324897896       100.8s   3008384241       1.105  33.0MHz   R3000

      444 cycles due to code that could not be
        assigned to any source procedure.


  621992586: Total number of Load Instructions executed.
 2468594083: Total number of bytes loaded by the program.
   62697078: Total number of Store Instructions executed.
  238107014: Total number of bytes stored by the program.


   39457261: Total number nops executed in branch delay slot.
  797923655: Total number conditional branches executed.
  298262224: Total number conditional branches actually taken.
          0: Total number conditional branch likely executed.
          0: Total number conditional branch likely actually taken.


  812198388: Total cycles waiting for current instr to finish.
 1674566430: Total cycles lost to satisfy scheduling constraints.
  719548954: Total cycles lost waiting for operands be available.


------------------------------------------------------------------------
*  -p[rocedures] using basic-block counts.                              *
*     Sorted in descending order by the number of cycles executed in each    *
*     procedure. Unexecuted procedures are not listed.                 *
------------------------------------------------------------------------

      cycles(%)  cum %     secs    instrns    calls procedure(file)

1615582437(48.59)          48.59                    48.96      1615582437      230795539
__mp_slave_wait_for_work(pvg-V8:mp_parallel_do.s)
1445167406(43.47)  92.06    43.79 1138163590      480 zigzag(pvg-V8:zigzag.c)
 133244569( 4.01)  96.06     4.04  123806971  1048622 fread(/usr/lib/libc.so.1:fread.c)
  52925160( 1.59)  97.65     1.60   52925160  3916800 hitgrnd(pvg-V8:hitgrnd.c)
```

```
 38804983( 1.17) 98.82    1.18  38804983  1048729 memcpy(/usr/lib/libc.so.1:bcopy.s)
     12780960(   0.38)      99.21              0.39          12780960                    60
gl_mem_hton_long_array(/usr/lib/libgl.so:../dgl/memory.c)
 12661888( 0.38) 99.59    0.38  12661888      16 get_block(pvg-V8:get_block.c)
 10840769( 0.33) 99.91    0.33  10840769      61 mp_wait_for_loop_completion(pvg-V8:mp.c)
  258048( 0.01) 99.92    0.01    258048    4096 fseek(/usr/lib/libc.so.1:fseek.c)
  246701( 0.01) 99.93    0.01    246701    4112 __filbuf(/usr/lib/libc.so.1:_filbuf.c)
  205636( 0.01) 99.93    0.01    205636    8659 strpbrk(/usr/lib/libc.so.1:strpbrk.c)
  185880( 0.01) 99.94    0.01    137460     300 matnmul(pvg-V8:matnmul.c)
  159759( 0.00) 99.94    0.00    159759       3 _bfclr(/usr/lib/libc.so.1:bit.c)
          151561(   0.00)      99.95              0.00              151561                3840
__mp_enter_independent(pvg-V8:mpc_interface.c)
  119821( 0.00) 99.95    0.00    119206     615 _bftstset(/usr/lib/libc.so.1:bit.c)
          111360(   0.00)      99.96              0.00              111360                3840
__mp_construct_exit_processing(pvg-V8:mp_pdo.c)
  107158( 0.00) 99.96    0.00    107158     262 _usinitsema(/usr/lib/libc.so.1:usemas.c)
   50267( 0.00) 99.96    0.00     50267    1169 gl_comm_io(/usr/lib/libgl.so:../dgl/comm.c)
   45632( 0.00) 99.96    0.00     45632     736 comm_fillbuffer(/usr/lib/libgl.so:../dgl/comm.c)
   45245( 0.00) 99.96    0.00     45245     179 _bzero(/usr/lib/libc.so.1:bzero.s)
   42240( 0.00) 99.97    0.00     42240     480 __mpu_main_1(pvg-V8:pvg-main.c)
   41516( 0.00) 99.97    0.00     41516     428 gl_comm_flush(/usr/lib/libgl.so:../dgl/comm.c)
   36840( 0.00) 99.97    0.00     36840     167 _memccpy(/usr/lib/libc.so.1:memccpy.c)
   36547( 0.00) 99.97    0.00     36547     689 _bcopy(/usr/lib/libc.so.1:bcopy.s)
           34960(   0.00)      99.97              0.00               34960                 368
gl_comm_read_data(/usr/lib/libgl.so:../dgl/comm.c)
   33740( 0.00) 99.97    0.00     32794     272 _lmalloc(/usr/lib/libc.so.1:amalloc.c)
   32667( 0.00) 99.97    0.00     32667     615 _hnewlock(/usr/lib/libc.so.1:hlocks.c)
   29304( 0.00) 99.97    0.00     29304    4875 _read(/usr/lib/libc.so.1:read.s)
   29200( 0.00) 99.97    0.00     23674      80 _inet_aton(/usr/lib/libc.so.1:inet_addr.c)
   27244( 0.00) 99.97    0.00     27244       7 mp_slave_control(pvg-V8:mp.c)
   26857( 0.00) 99.98    0.00     26857     961 _hsetlock(/usr/lib/libc.so.1:hlocks.c)
   25380( 0.00) 99.98    0.00     19080     180 __sin(/usr/lib/libm.so:trig.s)
   24702( 0.00) 99.98    0.00     24702    4117 _lseek(/usr/lib/libc.so.1:lseek.s)
   23912( 0.00) 99.98    0.00     23912     427 mp_slave_wait_for_work(pvg-V8:mp.c)
   21142( 0.00) 99.98    0.00     21142      62 clntudp_call(/usr/lib/libc.so.1:clnt_udp.c)
   19841( 0.00) 99.98    0.00     19841     349 __malloc(/usr/lib/libc.so.1:malloc.c)
   18681( 0.00) 99.98    0.00     18681    4395 strlen(/usr/lib/libc.so.1:strlen.s)
   18540( 0.00) 99.98    0.00     18540      60 __mp_fork(pvg-V8:mp.c)
   16745( 0.00) 99.98    0.00     16745      36 _doprnt(/usr/lib/libc.so.1:doprnt.c)
   16011( 0.00) 99.98    0.00     16011     305 cleanfree(/usr/lib/libc.so.1:malloc.c)
>
```

2. The version to be profiled is PVG without mouse.

```
> prof -pixie pvg-nomoV8 pvg-nomoV8.Counts.*
-----------------------------------------------------------------------------
Profile listing generated Fri Sep  8 13:20:22 1995
     with:          prof -pixie pvg-nomoV8 pvg-nomoV8.Counts.15870 pvg-nomoV8.Counts.15873
pvg-nomoV8.Counts.15874        pvg-nomoV8.Counts.15875        pvg-nomoV8.Counts.15876
pvg-nomoV8.Counts.15877 pvg-nomoV8.Counts.15878 pvg-nomoV8.Counts.15879
-----------------------------------------------------------------------------
```

```
Total cycles   Total Time  Instructions Cycles/inst   Clock   Target
 3068513132       92.99s   2816049716      1.090  33.0MHz   R3000

      320 cycles due to code that could not be
        assigned to any source procedure.


 612544507: Total number of Load Instructions executed.
2430930546: Total number of bytes loaded by the program.
 50134388: Total number of Store Instructions executed.
187903113: Total number of bytes stored by the program.


 31950871: Total number nops executed in branch delay slot.
816082753: Total number conditional branches executed.
297838296: Total number conditional branches actually taken.
      0: Total number conditional branch likely executed.
      0: Total number conditional branch likely actually taken.


828438751: Total cycles waiting for current instr to finish.
1389720414: Total cycles lost to satisfy scheduling constraints.
570565998: Total cycles lost waiting for operands be available.


-----------------------------------------------------------------------
*  -p[rocedures] using basic-block counts.                              *
*    Sorted in descending order by the number of cycles executed in each    *
*    procedure. Unexecuted procedures are not listed.                     *
-----------------------------------------------------------------------

     cycles(%)  cum %    secs    instrns    calls procedure(file)

1705860443(55.59)          55.59                  51.69     1705860443     243692397
__mp_slave_wait_for_work(pvg-nomoV8:mp_parallel_do.s)
1136754152(37.05)  92.64    34.45  893735492      182 zigzag(pvg-nomoV8:zigzag.c)
 133244569( 4.34)  96.98     4.04  123806971  1048622 fread(/usr/lib/libc.so.1:fread.c)
  39635306( 1.29)  98.27     1.20   39635306  2980608 hitgrnd(pvg-nomoV8:hitgrnd.c)
  38803952( 1.26)  99.54     1.18   38803952  1048724 memcpy(/usr/lib/libc.so.1:bcopy.s)
  12661888( 0.41)  99.95     0.38   12661888       16 get_block(pvg-nomoV8:get_block.c)
   258048( 0.01)  99.96     0.01     258048     4096 fseek(/usr/lib/libc.so.1:fseek.c)
   246701( 0.01)  99.97     0.01     246701     4112 __filbuf(/usr/lib/libc.so.1:_filbuf.c)
   205636( 0.01)  99.97     0.01     205636     8659 strpbrk(/usr/lib/libc.so.1:strpbrk.c)
            66060(   0.00)      99.97              0.00                 66060              1680
__mp_enter_independent(pvg-nomoV8:mpc_interface.c)
            48720(   0.00)      99.98              0.00                 48720              1680
__mp_construct_exit_processing(pvg-nomoV8:mp_pdo.c)
    36840( 0.00)  99.98     0.00     36840      167 _memccpy(/usr/lib/libc.so.1:memccpy.c)
    33574( 0.00)  99.98     0.00     33574      688 _bcopy(/usr/lib/libc.so.1:bcopy.s)
    29200( 0.00)  99.98     0.00     23674       80 _inet_aton(/usr/lib/libc.so.1:inet_addr.c)
    27244( 0.00)  99.98     0.00     27244        7 mp_slave_control(pvg-nomoV8:mp.c)
    24939( 0.00)  99.98     0.00     24939     4149 _read(/usr/lib/libc.so.1:read.s)
```

```
 24684( 0.00)  99.98    0.00    24684    4114 _lseek(/usr/lib/libc.so.1:lseek.s)
 23912( 0.00)  99.98    0.00    23912     427 mp_slave_wait_for_work(pvg-nomoV8:mp.c)
 21142( 0.00)  99.98    0.00    21142      62 clntudp_call(/usr/lib/libc.so.1:clnt_udp.c)
 20426( 0.00)  99.98    0.00    20426     420 __mpu_main_1(pvg-nomoV8:pvg-main.c)
 19050( 0.00)  99.98    0.00    19050     337 __malloc(/usr/lib/libc.so.1:malloc.c)
 17965( 0.00)  99.99    0.00    17965    4225 strlen(/usr/lib/libc.so.1:strlen.s)
 15899( 0.00)  99.99    0.00    15899      33 _doprnt(/usr/lib/libc.so.1:doprnt.c)
 15886( 0.00)  99.99    0.00    15886     357 _xdr_bytes(/usr/lib/libc.so.1:xdr.c)
 15536( 0.00)  99.99    0.00    15536     294 cleanfree(/usr/lib/libc.so.1:malloc.c)
 15354( 0.00)  99.99    0.00    15354     326 realfree(/usr/lib/libc.so.1:malloc.c)
 15111( 0.00)  99.99    0.00    15111     171 fgets(/usr/lib/libc.so.1:fgets.c)
 14936( 0.00)  99.99    0.00    14936     677 _xdr_u_long(/usr/lib/libc.so.1:xdr.c)
 14637( 0.00)  99.99    0.00    14637      80 _gethtent(/usr/lib/libc.so.1:gethostnam.c)
 12334( 0.00)  99.99    0.00    12334      70 _hsetlock(/usr/lib/libc.so.1:hlocks.c)
 11589( 0.00)  99.99    0.00    11589     296 _xdr_opaque(/usr/lib/libc.so.1:xdr.c)
 11460( 0.00)  99.99    0.00    11460      57 interpret(/usr/lib/libc.so.1:getservent.c)
 10433( 0.00)  99.99    0.00    10433      54 fflush(/usr/lib/libc.so.1:flush.c)
 10266( 0.00)  99.99    0.00    10266     354 xdrmem_putbytes(/usr/lib/libc.so.1:xdr_mem.c)
  9611( 0.00)  99.99    0.00     9611       7 clean_tidmap(/usr/lib/libc.so.1:usinit.c)
  8816( 0.00)  99.99    0.00     8816     551 xdrmem_getlong(/usr/lib/libc.so.1:xdr_mem.c)
  8658( 0.00)  99.99    0.00     8658     333 _malloc(/usr/lib/libc.so.1:malloc.c)
  8586( 0.00)  99.99    0.00     8586     318 __free(/usr/lib/libc.so.1:malloc.c)
  8340( 0.00)  99.99    0.00     8340      12 _dn_comp(/usr/lib/libc.so.1:res_comp.c)
  7950( 0.00)  99.99    0.00     7950     318 _free(/usr/lib/libc.so.1:malloc.c)
>
```

3. The last version to be profiled will be PVG with no video.

```
> prof -pixie pvg-nomoV8 pvg-noV8.Counts.*
------------------------------------------------------------------------------
Profile listing generated Fri Sep  8 13:27:03 1995
      with:          prof -pixie pvg-nomoV8 pvg-noV8.Counts.15927 pvg-noV8.Counts.15929
pvg-noV8.Counts.15930 pvg-noV8.Counts.15931 pvg-noV8.Counts.15932 pvg-noV8.Counts.15933
pvg-noV8.Counts.15934 pvg-noV8.Counts.15935
------------------------------------------------------------------------------

Total cycles   Total Time  Instructions Cycles/inst   Clock   Target
 1809960505       54.85s    1479384732       1.223  33.0MHz   R3000

        116 cycles due to code that could not be
          assigned to any source procedure.


  174391084: Total number of Load Instructions executed.
  678397908: Total number of bytes loaded by the program.
   58645832: Total number of Store Instructions executed.
  221939742: Total number of bytes stored by the program.


   40767816: Total number nops executed in branch delay slot.
  132662461: Total number conditional branches executed.
   76340487: Total number conditional branches actually taken.
```

```
        0: Total number conditional branch likely executed.
        0: Total number conditional branch likely actually taken.


   146891144: Total cycles waiting for current instr to finish.
  1490471589: Total cycles lost to satisfy scheduling constraints.
   748192510: Total cycles lost waiting for operands be available.


------------------------------------------------------------------------
*  -p[rocedures] using basic-block counts.                             *
*     Sorted in descending order by the number of cycles executed in each    *
*     procedure. Unexecuted procedures are not listed.                 *
------------------------------------------------------------------------

      cycles(%)  cum %    secs    instrns    calls procedure(file)

1500623520(82.91)  82.91    45.47 1179547920      480 zigzag(pvg-noV8:zigzag.c)
 133238736( 7.36)  90.27     4.04  123801552  1048576 fread(/usr/lib/libc.so.1:fread.c)
      61837496(    3.42)       93.69                  1.87          61837496        8831976
__mp_slave_wait_for_work(pvg-noV8:mp_parallel_do.s)
  51064500( 2.82)  96.51     1.55   51064500  3916800 hitgrnd(pvg-noV8:hitgrnd.c)
  38800610( 2.14)  98.65     1.18   38800610  1048632 memcpy(/usr/lib/libc.so.1:bcopy.s)
  12661888( 0.70)  99.35     0.38   12661888       16 get_block(pvg-noV8:get_block.c)
   9809010( 0.54)  99.89     0.30    9809010       61 mp_wait_for_loop_completion(pvg-noV8:mp.c)
    258048( 0.01)  99.91     0.01     258048     4096 fseek(/usr/lib/libc.so.1:fseek.c)
    246031( 0.01)  99.92     0.01     246031     4100 __filbuf(/usr/lib/libc.so.1:_filbuf.c)
    185880( 0.01)  99.93     0.01     137460      300 matnmul(pvg-noV8:matnmul.c)
    159759( 0.01)  99.94     0.00     159759        3 _bfclr(/usr/lib/libc.so.1:bit.c)
          155226(    0.01)       99.95                  0.00          155226        3840
__mp_enter_independent(pvg-noV8:mpc_interface.c)
    119821( 0.01)  99.96     0.00     119206      615 _bftstset(/usr/lib/libc.so.1:bit.c)
          111360(    0.01)       99.96                  0.00          111360        3840
__mp_construct_exit_processing(pvg-noV8:mp_pdo.c)
    107158( 0.01)  99.97     0.00     107158      262 _usinitsema(/usr/lib/libc.so.1:usemas.c)
     42240( 0.00)  99.97     0.00      42240      480 __mpu_main_1(pvg-noV8:pvg-main.c)
     38783( 0.00)  99.97     0.00      38783       71 _bzero(/usr/lib/libc.so.1:bzero.s)
     33740( 0.00)  99.97     0.00      32794      272 _lmalloc(/usr/lib/libc.so.1:amalloc.c)
     32667( 0.00)  99.98     0.00      32667      615 _hnewlock(/usr/lib/libc.so.1:hlocks.c)
     32437( 0.00)  99.98     0.00      32437      960 _hsetlock(/usr/lib/libc.so.1:hlocks.c)
     27244( 0.00)  99.98     0.00      27244        7 mp_slave_control(pvg-noV8:mp.c)
     25380( 0.00)  99.98     0.00      19080      180 __sin(/usr/lib/libm.so:trig.s)
     24690( 0.00)  99.98     0.00      24690     4115 _lseek(/usr/lib/libc.so.1:lseek.s)
     24618( 0.00)  99.98     0.00      24618     4103 _read(/usr/lib/libc.so.1:read.s)
     23912( 0.00)  99.98     0.00      23912      427 mp_slave_wait_for_work(pvg-noV8:mp.c)
     18540( 0.00)  99.99     0.00      18540       60 __mp_fork(pvg-noV8:mp.c)
     15320( 0.00)  99.99     0.00      15320       60 __mpc_region(pvg-noV8:mpc_interface.c)
     13100( 0.00)  99.99     0.00      13100      262 _usnewsema(/usr/lib/libc.so.1:usemas.c)
     13020( 0.00)  99.99     0.00      10140       60 genddcos(pvg-noV8:genddcos.c)
     12245( 0.00)  99.99     0.00      12245        9 clean_tidmap(/usr/lib/libc.so.1:usinit.c)
     12240( 0.00)  99.99     0.00       8760       60 veh2utm_matrix(pvg-noV8:veh2utm_matrix.c)
     12126( 0.00)  99.99     0.00      12126      267 _amalloc(/usr/lib/libc.so.1:amalloc.c)
```

```
10837( 0.00)  99.99    0.00     10837      33 getenv(/usr/lib/libc.so.1:getenv.c)
          10080(   0.00)      99.99            0.00                10080                480
__mp_exit_independent(pvg-noV8:mpc_interface.c)
   9852( 0.00)  99.99    0.00      9852      44 fflush(/usr/lib/libc.so.1:flush.c)
   9005( 0.00)  99.99    0.00      9005     846 nvmatch(/usr/lib/libc.so.1:getenv.c)
   8593( 0.00)  99.99    0.00      8593      22 _doprnt(/usr/lib/libc.so.1:doprnt.c)
   8058( 0.00)  99.99    0.00      8058    1030 _hcsetlock(/usr/lib/libc.so.1:hlocks.c)
   7976( 0.00)  99.99    0.00      7976       1 mp_create(pvg-noV8:mp.c)
   7770( 0.00)  99.99    0.00      7770     259 _usctlsema(/usr/lib/libc.so.1:usemas.c)
>
```

C. LOG91

1. This session will be conducted on alioth to get frame rates using the PVG stripped of video and limited to 60 frames. All times are in seconds.

1. 69.34
2. 68.34
3. 69.16
4. 68.36
5. 70.74
6. 69.23
7. 68.78
8. 70.91
9. 70.28
10. 68.34
11. 69.17
12. 69.47
13. 68.78
14. 70.17
15. 68.66
16. 68.43
17. 69.65
18. 68.15
19. 68.80
20. 68.78
21. 68.78
22. 71.34
23. 68.80
24. 69.40
25. 69.71
26. 71.16
27. 68.42
28. 69.34
29. 68.80
30. 69.65

2. This session will use the same version of PVG as above, except that pvg-main.c has been converted to an eight thread multi-processor porgram.

1. 12.62
2. 11.78
3. 11.96

4.	13.03
5.	13.34
6.	12.81
7.	12.90
8.	13.17
9.	12.84
10.	13.28
11.	13.09
12.	12.90
13.	12.81
14.	13.03
15.	12.71
16.	12.53
17.	12.55
18.	12.53
19.	11.84
20.	12.21
21.	12.96
22.	13.96
23.	12.42
24.	12.18
25.	11.71
26.	11.66
27.	11.84
28.	11.72
29.	11.71
30	12.22

D. LOG95(ALIOTH)

1. The first set of trials will establish a serial baseline for the PVG program with 60 frames of video. All times in seconds.

1. 54.10.
2. 54.15.
3. 54.24
4. 54.16
5. 54.03
6. 54.09
7. 54.09
8. 54.22
9. 53.90
10. 53.99
11. 53.84
12. 54.09
13. 53.85
14. 54.53
15. 53.90
16. 54.11
17. 54.46
18. 54.24
19. 53.86
20. 54.59
21. 54.28
22. 54.09
23. 54.11
24. 54.23
25. 54.11
26. 54.28
27. 53.87
28. 53.96
29. 54.03
30. 54.28

2. The next set of trials is on the same version of the PVG as above converted to an eight thread multi-processor version in pvg-main.c. CPU activity appears to be very low as viewed on gr_osview.

1. 10.47
2. 10.36

3. 10.60
4. 10.37
5. 10.78
6. 10.80
7. 10.16
8. 10.32
9. 10.09
10. 10.28
11. 10.78
12. 10.78
13. 10.65
14. 10.71
15. 10.53
16. 10.53
17. 10.48
18. 10.65
19. 10.34
20. 10.34
21. 10.30
22. 10.72
23. 10.40
24. 10.87
25. 10.46
26. 10.40
27. 10.59
28. 10.53
29. 10.32
30. 10.29

E. LOG96(ALIOTH)

1. This series of PVG testing employs the video version with no targets or overhead dsiplay. The mouse has been restored in order to judge the "smoothness"
of movement. As usual, a serila baseline is being dine first.

```
> who
ferret    ttyq0      Sep  5 19:15
frazier   ttyq1      Sep  4 22:29
dbmarco   ttyq2      Sep  5 19:26
jpbuziak  ttyq3      Sep  5 19:18
```

Inspite of the number of people logged on, cpu activity is light.

1. 53.28
2. 52.46
3. 52.15
4. 52.42
5. 53.96
6. 53.90
7. 53.84
8. 54.59
9. 54.59
10. 54.53
11. 54.84
12. 54.04
13. 54.28
14. 54.15
15. 53.42
16. 54.53
17. 54.65
18. 54.54
19. 53.67
20. 53.96
21. 53.67
22. 53.21
23. 54.74
24. 52.60
25. 54.68
26. 54.34
27. 54.73

28. 54.50
29. 54.66
30. 54.65

who
ferret ttyq0 Sep 5 19:15
frazier ttyq1 Sep 4 22:29
jpbuziak ttyq3 Sep 5 19:18

2. The second phase of testing will be on the same version of the PVG as above.
Except pvg-main.c has been converted to an eight thread multiprocessing program.
CPU activity is slightly higher than when starting the previous run.

1. 13.09
2. 11.84
3. 11.75
4. 12.28
5. 12.24
6. 12.03
7. 12.09
8. 13.43
9. 13.46
10. 12.10
11. 10.29
12. 12.87
13. 10.46
14. 10.78
15. 12.22
16. 12.24
17. 11.72
18. 11.85
19. 11.78
20. 11.41
21. 11.86
22. 10.68
23. 12.62
24. 12.34
25. 12.03
26. 12.12
27. 12.78

28. 11.90
29. 12.11
30 12.10

3. The next series of tests is with the profiler invoked. The purpose of this testing is to spot the next procedure to move in the parallel region.

```
> prof pvg-main
------------------------------------------------------------------------
Profile listing generated Tue Sep  5 20:51:59 1995
   with:      prof pvg-main
------------------------------------------------------------------------

samples  time   CPU    FPU   Clock  N-cpu S-interval Countsize
  1706    17s R2000A/R3000 R2010A/R3010  33.0MHz  0     10.0ms
0(bytes)

Each sample covers 4 bytes for every 10.0ms ( 0.06% of 17.0600sec)


------------------------------------------------------------------------
 -p[rocedures] using pc-sampling.
 Sorted in descending order by the number of samples in each procedure.
 Unexecuted procedures are excluded.
------------------------------------------------------------------------

samples   time(%)      cum time(%)     procedure (file)

    669   6.7s( 39.2)  6.7s( 39.2)       _read (/usr/lib/libc.so.1:read.s)
    607   6.1s( 35.6)   13s( 74.8)      fread (/usr/lib/libc.so.1:fread.c)
    233   2.3s( 13.7)   15s( 88.5)  get_block (pvg-main:get_block.c)
    125   1.2s(  7.3)   16s( 95.8)     memcpy (/usr/lib/libc.so.1:bcopy.s)
     19  0.19s(  1.1)   17s( 96.9)      _lseek (/usr/lib/libc.so.1:lseek.s)
     12  0.12s(  0.7)   17s( 97.6)       _open (/usr/lib/libc.so.1:open.s)
     10   0.1s(  0.6)   17s( 98.2)    __filbuf (/usr/lib/libc.so.1:_filbuf.c)
      6  0.06s(  0.4)   17s( 98.5)      _write (/usr/lib/libc.so.1:write.s)
      5  0.05s(  0.3)   17s( 98.8)      _close (/usr/lib/libc.so.1:close.s)
      4  0.04s(  0.2)   17s( 99.1)      fseek (/usr/lib/libc.so.1:fseek.c)
      4  0.04s(  0.2)   17s( 99.3)     _select (/usr/lib/libc.so.1:select.s)
      3  0.03s(  0.2)   17s( 99.5)       _fork (/usr/lib/libc.so.1:fork.s)
      1  0.01s(  0.1)   17s( 99.5)     _writev (/usr/lib/libc.so.1:writev.s)
      1  0.01s(  0.1)   17s( 99.6)  cleanfree (/usr/lib/libc.so.1:malloc.c)
```

```
     1  0.01s(  0.1)   17s( 99.6)        _XRead (/usr/lib/libX11.so.1:XlibInt.c)
     1  0.01s(  0.1)   17s( 99.7)       _recvmsg (/usr/lib/libc.so.1:recvmsg.s)
               1      0.01s(     0.1)       17s(  99.8)  _GLmakedefaultpalette
(/usr/lib/libgl.so:../glws/mex_colormap.c)
     1  0.01s(  0.1)   17s( 99.8)       _connect (/usr/lib/libc.so.1:connect.s)
     1  0.01s(  0.1)   17s( 99.9) XextFindDisplay (/usr/lib/libXext.so:extutil.c)
               1      0.01s(     0.1)       17s(  99.9)          _BSD_getime
(/usr/lib/libc.so.1:BSD_getime.s)
     1  0.01s(  0.1)   17s(100.0)         XSync (/usr/lib/libX11.so.1:Sync.c)

  1706    17s(100.0)   17s(100.0)         TOTAL
```

4. The profile below was taken on the ME system.

```
prof pvg-main
------------------------------------------------------------------------
Profile listing generated Wed Sep  6 10:50:51 1995
   with:       prof pvg-main
------------------------------------------------------------------------

samples   time    CPU    FPU   Clock   N-cpu  S-interval Countsize
   794   7.9s  R4400  R4010 100.0MHz   1      10.0ms     2(bytes)

Each sample covers 4 bytes for every 10.0ms ( 0.13% of 7.9400sec)


------------------------------------------------------------------------
  -p[rocedures] using pc-sampling.
  Sorted in descending order by the number of samples in each procedure.
  Unexecuted procedures are excluded.
------------------------------------------------------------------------

samples   time(%)      cum time(%)      procedure (file)

    409   4.1s( 51.5)  4.1s( 51.5)        _read (libc.so.1:read.s)
    226   2.3s( 28.5)  6.3s( 80.0)        fread (libc.so.1:fread.c)
     80   0.8s( 10.1)  7.1s( 90.1)    get_block (pvg-main:get_block.c)
     59  0.59s(  7.4)  7.7s( 97.5)      memcpy (libc.so.1:bcopy.s)
      6  0.06s(  0.8)  7.8s( 98.2)       _close (libc.so.1:close.s)
      4  0.04s(  0.5)  7.8s( 98.7)        _open (libc.so.1:open.s)
      4  0.04s(  0.5)  7.9s( 99.2)       _lseek (libc.so.1:lseek.s)
```

```
  3  0.03s(  0.4)  7.9s( 99.6)     __filbuf (libc.so.1:_filbuf.c)
  1  0.01s(  0.1)  7.9s( 99.7)      _fork (libc.so.1:fork.s)
  1  0.01s(  0.1)  7.9s( 99.9)     fseek (libc.so.1:fseek.c)
  1  0.01s(  0.1)  7.9s(100.0)    nvmatch (libc.so.1:getenv.c)

794  7.9s(100.0)  7.9s(100.0)       TOTAL
```

F. LOG98(ALIOTH)

1. This series of trials will compare diferent versions of the PVG algorithm running in 8 wide multi-processor threads. Testing will begin with pvg, no video. CPU actvity is moderate based on observing gr_osview.

1. 11.9
2. 10.35
3. 10.54
4. 12.23 Major Operating System Activity.
5. 10.37
6. 9.28
7. 9.65
8. 9.60
9. 9.41
10. 9.40
11. 9.46
12. 9.46
13. 9.37
14. 9.71
15. 9.31
16. 9.18
17. 9.43
18. 9.09
19. 9.42
20. 9.36
21. 9.28
22. 9.17
23. 9.34
24. 9.41
25. 9.56
26. 9.35
27. 9.53
28. 9.67
29. 9.84
30. 9.40

2. The next set of trials is with video, but without mouse.

1. 18.84
2. 18.15
3. 18.90

4. 18.54
5. 18.00
6. 18.42
7. 18.29
8. 18.31
9. 18.78
10. 18.09
11. 18.47
12. 18.65
13. 19.11
14. 18.30
15. 18.65
16. 18.55
17. 18.42
18. 18.62
19. 18.02
20. 18.59
21. 18.15
22. 18.15
23. 18.15
24. 18.28
25. 18.61
26. 18.43
27. 18.04
28. 18.36
29. 17.78
30. 19.15

3. The last series of tests will be on the PVG with mouse.

1. 12.66
2. 13.12
3. 12.91
4. 12.46
5. 14.68
6. 13.09
7. 13.98
8. 12.91
9. 12.46
10. 12.78
11. 13.06
12. 12.84

13. 12.56
14. 10.28
15. 11.78
16. 11.17
17. 11.79
18. 11.23
19. 11.85
20. 13.12
21. 11.92
22. 10.65
23. 10.68
24. 11.78
25. 11.00
26. 10.65
27. 11.78
28. 13.09
29. 12.06
30. 12.17

G. LOG925(TOBAGO)

1. The first trial will be on the serial version of the PVG. System activity is low to non-exsistent. All times in seconds.

1. 17.13
2. 18.03
3. 17.22
4. 18.15
5. 17.61
6. 18.03
7. 18.46
8. 18.53
9. 17.96
10. 18.04
11. 17.18
12. 17.85
13. 17.59
14. 18.28
15. 17.84
16. 17.80
17. 18.11
18. 17.90
19. 18.21
20. 17.80
21. 18.24
22. 17.78
23. 17.47
24. 18.00
25. 18.29
26. 17.11
27. 18.56
28. 17.84
29. 17.60
30. 18.15

2. The next set of trials will be on the PVG algorithm without video. All times are in seconds.

1. 5.67
2. 5.30

3. 5.49
4. 5.53
5. 5.59
6. 5.59
7. 5.54
8. 5.72
9. 5.66
10. 5.46
11. 5.53
12. 5.46
13. 4.96
14. 5.53
15. 5.48
16. 5.43
17. 5.65
18. 5.53
19. 5.59
20. 5.48
21. 5.46
22. 5.86
23. 5.59
24. 5.85
25. 5.55
26. 5.60
27. 5.49
28. 5.24
29. 5.48
30. 5.46

3. The next trial will be on the PVG with the video, but without any input from the mouse. All times in seconds.

1. 6.40
2. 6.30
3. 6.35
4. 6.15
5. 6.84
6. 6.51
7. 6.40
8. 6.37
9. 6.28
10. 6.23

11. 6.37
12. 6.12
13. 6.40
14. 6.21
15. 6.28
16. 6.24
17. 6.28
18. 6.15
19. 6.49
20. 6.31
21. 6.40
22. 6.18
23. 6.36
24. 6.40
25. 6.21
26. 6.21
27. 6.34
28. 6.34
29. 6.40
30. 6.18

4. The last trial will be conducted on the full PVG.
All times in seconds.

1. 6.53
2. 6.28
3. 6.71
4. 6.90
5. 6.67
6. 6.62
7. 6.59
8. 6.80
9. 6.53
10. 5.61
11. 6.47
12. 6.84
13. 5.78
14. 6.78
15. 6.56
16. 6.21
17. 6.78
18. 6.80

19. 6.71
20. 6.53
21. 6.84
22. 6.66
23. 6.37
24. 7.21
25. 6.67
26. 7.05
27. 6.84
28. 6.46
29. 6.84
30. 5.96

H. LOG121 (ALGIEBA)

1. Serial Trial on Algieba with all the features active.
60 Frame tirals. All times in seconds.

1. 15.96
2. 16.28
3. 16.68
4. 16.62
5. 17.37
6. 16.49
7. 16.84
8. 16.71
9. 16.96
10. 17.16
11. 17.36
12. 17.11
13. 16.34
14. 16.67
15. 16.46
16. 16.84
17. 16.98
18. 16.46
19. 16.56
20. 16.68
21. 16.84
22. 16.46
23. 16.84
24. 16.92
25. 16.71
26. 16.73
27. 17.03
28. 17.09
29. 16.96
30. 16.90

2. Parallel Trial on Algieba with all features active. 60 Frame trial, all times in seconds.

1. 10.03
2. 11.81
3. 10.40

4. 9.69
5. 10.40
6. 10.31
7. 10.28
8. 10.49
9. 10.30
10. 10.30
11. 10.80
12. 10.43
13. 10.80
14. 9.78
15. 10.92
16. 10.71
17. 10.21
18. 10.00
19. 10.40
20. 10.09
21. 10.05
22. 10.19
23. 9.90
24. 10.15
25. 10.68
26. 10.25
27. 10.09
28. 10.30
29. 9.75
30. 9.90

APPENDIX D

A. TEST PROCEDURE NUMBER ONE

Purpose:

To provide a standard procedure for testing and comparing C computer programs compiled for serial and parallel execution.

Equipment:

The programs will be run on two architecturally different computers: Alioth (SGI 4D/380 with 8 R2000 processors) and Algieba (SGI Power Onyx VTX with 2 R4000 processors). Both computers employ an SGI Unix Version 5 operating system. Each is also equipped with the Iris Power C Analyzer(PCA) Version 2.4.2.

Procedure:

Unix commands to be executed at the keyboard will be denoted using the following font: ***command.*** Steps 1 through 12 will only be accomplished once at the beginning of testing to provide a baseline for comparison. The sequence of steps 13-21 need only be executed once to obtain a baseline for comparison later. The sequence of steps 22- 30 will be repeated for each feature of the PCA demonstrated.

1. Programs will be written to the ANSI C++ standard and reside in the root directory of the test account.
2. A jot session will be opened and used to record all pertinent observations and any error messages or results not saved to a separate file.
3. The command ***who*** will executed. Using the "drag and drop" feature of the X Window Graphical User interface (GUI) the result will be copied to the jot window.
4. The ***ls*** command will be executed to verify the presence of the source file in the root directory and ensure no output files from previous testing exist. If any unwanted files are present they should be eliminated using the ***rm*** command. Once the directory is clear and ready to proceed the ***ls*** command will be executed a final time. The results will be copied to the jot window using the "drag and drop" feature of the GUI.
5. The source file, or test function, will be compiled and executed using the profiling feature of the operating system. This is done using the command specified below.

cc -p testfn.c *-o* testfn
ls

6. The ***ls*** command is executed in order to verify the executable file was created. The
results of the directory list should be copied to thejot window as previously discussed.

7. The program will then be executed by entering the name of the executable file at the
keyboard and depressing enter.

8. An output file will be generated during execution; mon.out. The ***ls*** command should be executed to ensure the file was successfully generated.

9. A file documenting the results of the profiling can be generated using the command below. The file test? can be printed for review at a later date.

prof testfn > test?

10. The source file should now be recompiled without the profiling feature included in the executable file.

cc testfn.c **-o** testfn

11. The pixie feature can now be employed to further analyze the performance of the program. This is accomplished using the series of commands listed below.

pixie testfn
testfn.pixie
prof testfn **-pl-rie** > test??

12. The output file, test??, can be printed as a text file and analyzed at a later date.

13. The ***ls*** command will be executed to verify the presence of the source file in the root directory and ensure not output files from previous testing exist. If any unwanted files are present they should be eliminated using the ***rm*** command. Once the directory is clear and ready to proceed the **ls** command will be executed a final time. The results will be copied to the jot window using the "drag and drop" feature of the GUI.

14. The source file, or test function, will compiled and executed using the pca and profiling feature of the operating system. This is done using the commands specified below. The file pcafn?.c should be retained as it documents the manner in which PCA modifies the code for parallel execution.

usrlliblpca testfn.c > pcafn?.c
cc **-p** pcafn?.c **-o**
ls

15. The **ls** command is executed in order to verify the executable file was created. The

results of the directory list should be copied to the jot window as previously discussed.

16. The program will then be executed by entering the name of the executable file at the keyboard and depressing enter.

17. An output file will be generated during execution; mon.out. The **ls** command should be executed to ensure the file was successfully generated.

18. A file documenting the results of the profiling can be generated using the command below. The file test? can be printed for review at a later date.

prof pcafn? > test?

19. The source file should now be recompiled without the profiling feature included in the executable file.

c pcafn?.c **-o** pcafn?

20. The pixie feature can now be employed to further analyze the performance of the program. This is accomplished using the series of commands listed below.

pixie pcafn?
pcafn.pixie
prof pcafn? ***-pixie*** > test??

21. The output file, test??, can be printed as a text file and analyzed at a later date.

22. The ***ls*** command will be executed to verify the presence of the source file in the root directory and ensure not output files from previous testing exist. If any unwanted files are present they should be eliminated using the ***rm*** command. Once the directory is clear and ready to proceed the ***ls*** command will be executed a final time. The results will be copied to the jot window using the "drag and drop" feature of the GUI.

23. The source file, or test function, will compiled and executed using the pca with selected options and profiling feature of the operating system. This is done using the commands specified below. The file pcafn?.c should be retained as it documents the manner in which PCA modifies the code for parallel execution. The output file from the pca should be modified so that it is distinct from previous runs on the test function.

usr/lib/pca [options] testfn.c > pcafn?.c C ***c -p*** pcafn?.c **-o is**

24. The ***ls*** command is executed in order to verify the executable file was created. The results of the directory list should be copied to the jot window as previously discussed.

25. The program will then be executed by entering the name of the executable file at the keyboard and depressing enter.

26. An output file will be generated during execution; mon.out. The ***ls*** command should be executed to ensure the file was successfully generated.

27. A file documenting the results of the profiling can be generated using the command

below. The file test? can be printed for review at a later date.

prof pcafn? > test?

28. The source file should now be recompiled without the profiling feature included in the executable file.

cc pcafn?.c **-o** pcafn?

29. The pixie feature can now be employed to further analyze the performance of the program. This is accomplished using the series of commands listed below.

pixie pcafn?
pcafn.pixie
prof *pcafn? -pixie* > test??

30. The output file, test??, can be printed as a text file and analyzed at a later date.

B. TEST PROCEDURE NUMBER TWO

Purpose:

To provide a basis for comparison between two similar programs performing the same function with regards to the "speed" of a program. The speed of the program can be conceived of in two different ways. The most empirical analysis would come from determining the execution time of a program. Where there is significant input/output (I/O) associated with the program observed performance on the screen may differ significantly from the execution time. In this case manual timing using a stop watch may have to be resorted to. Given the random error which is associated with both cases, in excess of 30 or more trials should be conducted in order to ensure Gaussian distributions for the results.

Equipment:

Serial testing of the Perspective View Generator will be conducted on several machines; the Mechanical Engineering Departments SGI work station, Alioth and Algieba located in the Visualization Lab. Each of these systems utilizes an SGI Unix Version 5 operating system and have essentially the same features available.

A stop watch or similar timing device will be required for Phase 2.

Procedure:

The following testing should always be conducted at a station as electronically close to the target machine as possible. Remote operation over nets should be avoided. This is due to the fact that timing will require observations be made of the screen to determine start and stop times. Operation remotely would introduce additional variation in times.

For the following procedure UNIX commands will be highlighted using the following font: ***example***. Any other text, such as filenames, appearing on the command line will appear in a font consistent with the text. Where ever a '?' appears in a command line it should be treated as wild card. This typically represent a location where a serial is inserted to denote different trials or test runs.

Throughout the testing a ***jot*** window will be available to record all significant commands entered in the Unix Shell. Output not otherwise saved, and as noted in the procedure, will also be saved in the ***jot*** window. The tester is free to add comments to the ***jot*** window as appropriate throughout testing.

All programs tested will be written in ANSI C++ and reside in the test directory. Makefiles and related programs may have to be modified from trial to trial. Those modifications are not part of this procedure.

The phases which appear below will each be repeated thirty times for the reason indicated above.

Phase 1 - Code Profiling

1. Enter the ***ls*** command and observe that there are no files unconnected with testing, which can be removed, present in the directory. If any unwanted files are present use the ***rm***

command to remove them. When the directory is prepared copy the *ls* command and the resulting output in the jot window. This should include the command line prompt showing the name of the target machine.

2. Run the test code as indicated below.
3. Enter data as prompted by the program.
4. Depress the left mouse button to place the first window(Map View).
5. Depress the left mouse button to place the second window (Main View). Observe the main view carefully. The view should move in jerks, each jerk is a frame. Count the number of jerks.
6. As you are counting move the pointer to the upper left hand corner of the screen and place it over the icon block to close the window.
7. On reaching forty frames close the window. This terminates the program.
8. Run the profiler as indicated below. Copy the output to the *jot* window.

pvg-main
profile pvg-main

Note: The output only needs to copied to the *jot* window for the first trial. Thereafter only record the total run time which appears at the bottom of the output. This is acceptable since the proportion the program spends in each procedure is relatively invariant.

Phase 2 - Frame Rate Estimate

1. Run the program as indicated above with modifications as noted below.
2. Before depressing the left mouse button to place the Main Window have the stop watch in hand and ready to start.
3. Simultaneously start the second window and the stop watch.
4. Count 20 frames and stop the stopwatch.
5. Terminate the program by closing the second window.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center 8725 John J. Kingman Rd., STE 0944 Ft. Belvoir, Virginia 22060-6218	2
2.	Library, Code 13 Naval Postgraduate School Monterey, California 93943-5101	2
3.	Department Chairmen, Code ME Department of Mechanical Engineering Naval Postgraduate School Monterey, California 93943-5000	1
4.	Professor Morris E. Driels Code ME/Dr Department of Mechanical Engineering Naval Postgraduate School Monterey, California 93943-5000	2
5.	John P. Buziak 239 Olivier Street New Orleans, Louisiana 70114	2
6.	Naval Engineering Curricular Officer, Code 34 Department of Mechanical Engineering Naval Postgraduate School Monterey, California 93943-5000	1
7.	Director, TRAC-MTRY Box 8692 Naval Postgraduate School Monterey, California 93943	2
8.	Adj. Professor Judith Lind Code OR/Li Department of Operations Research Naval Postgraduate School Monterey, California 93943-5000	1

9. Adj. Professor Wolfgang Baer 1
Code CS/Ba
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943-5000